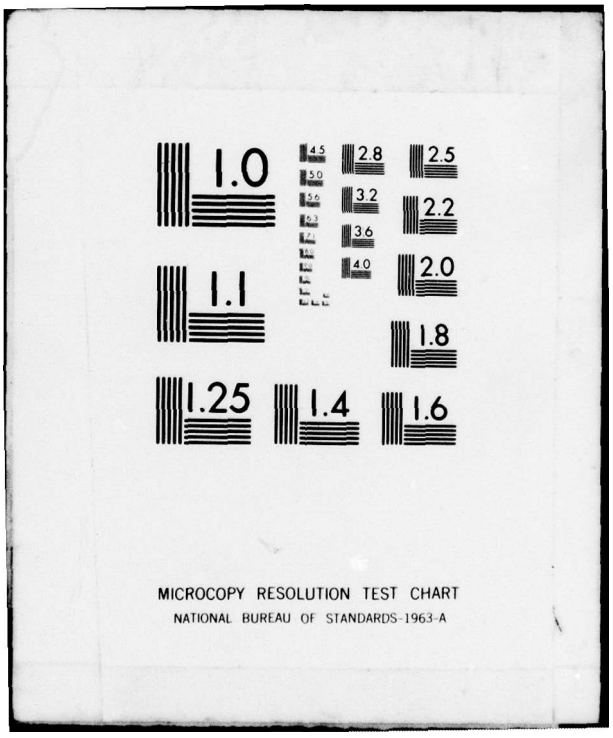
AD-A076 021 HONEYWELL SYSTEMS AND RESEARCH CENTER MINNEAPOLIS MN F/G 1/3
DIGITAL FLIGHT CONTROL SOFTWARE VALIDATION STUDY.(U)
JUN 79 E R RANG , M J GUTMANN , D B MULCARE F33615-78-C-3605
UNCLASSIFIED 79SRC18 AFFDL-TR-79-3076 NL

1 OF 3
AD
A076021


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AFFDL-TR-79-3076

LEVEL II

DIGITAL FLIGHT CONTROL SOFTWARE VALIDATION STUDY

E. R. Rang
M. J. Gutmann
D. B. Mulcare
W. G. Ness

Honeywell Systems and Research Center
2600 Ridgway Parkway
Minneapolis, Minnesota 55413

June 1979

Final Report for Period April 1978 to April 1979

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

AIR FORCE FLIGHT DYNAMICS LABORATORY
AIR FORCE WRIGHT AERONAUTICAL LABORATORIES
AIR FORCE SYSTEMS COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OHIO 45433

NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely related Government procurement operation, the United States Government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or corporation, or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

This report has been reviewed by the Information Office (OI) and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.

Gerald W. Francis
Project Manager

Duane P. Rubertus, Chief
Control Systems Development Branch
Flight Control Division

FOR THE COMMANDER

MORRIS A. OSTGAARD
Assistant for Research
and Technology
Flight Control Division

"If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization please notify AFFDL/FGL *,W-PAFB, OH 45433 to help us maintain a current mailing list".*

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

AIR FORCE/56780/24 September 1979 — 240

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: AFFDL-TR-79-3076	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER: Final rept.	
4. TITLE (and Subtitle): DIGITAL FLIGHT CONTROL SOFTWARE VALIDATION STUDY	5. TYPE OF REPORT & PERIOD COVERED: Technical Final, April 1978 to April 1979
	6. PERFORMING ORG. REPORT NUMBER: 79SRC18
7. AUTHOR(s): E.R. Rang, D.B. Mulcare, M.J. Gutmann, W.G. Ness	8. CONTRACT OR GRANT NUMBER(s): F33615-78-C-3605
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Honeywell Systems and Research Center, 2600 Ridgway Parkway, Minneapolis, MN 55413	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 2403-02-04
11. CONTROLLING OFFICE NAME AND ADDRESS: Air Force Flight Dynamics Laboratory (FGL), Air Force Systems Command, Wright-Patterson Air Force Base, Ohio 45433	12. REPORT DATE: June 1979
	13. NUMBER OF PAGES: 265
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): 12 2621	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMEN

Edward R. /Rang, Michael J. /Gutmann, Dennis B. /Mulcare William G. /Ness

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Flight controls
Embedded computer systems
Software verification
System validation
Software development
Digital autopilots

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The problems of designing, verifying, and validating software for digital flight control systems are reviewed to study how the new software engineering tools and techniques may be incorporated into the development process. This shows how automated methodologies will provide error-free flight control software at lower costs. The need for expensive, lengthy test programs is reduced by analytical methods. The quality of the software is demonstrated with higher confidence when designs are structured to facilitate the subsequent

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE When Data Entered

402349

20. Abstract (concluded)

verification steps. The military standards relating to flight control systems are reviewed, and modifications to include provisions for software verification are recommended.

PREFACE

This project, conducted jointly between the Honeywell Systems and Research Center and the Lockheed-Georgia Company, was monitored by Gerald W. Francis, AFFDL. The principal contributers were Michael Gutmann, Dennis Mulcare, William Ness, and Edward Rang. Earl Boebert, James Bores, Victor Falkner, and Thomas Lahn helped with data, details, and consultation. The study began in April 1978 and was concluded the following April.

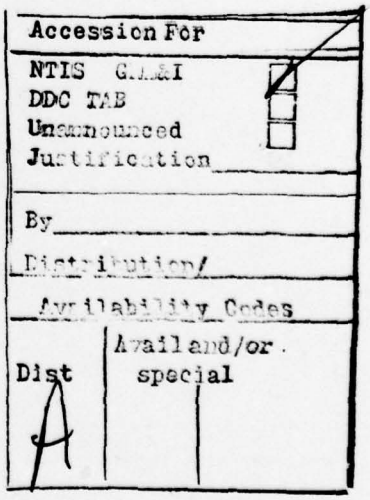


TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (concluded)

LIST OF ILLUSTRATIONS

LIST OF TABLES

LIST OF TABLES (concluded)

SUMMARY

The aim of this work is to outline the problems and methods in demonstrating computer programs for flight control systems are free of errors. This is to provide a foundation from which techniques for verifying the software at each stage of the development cycle may be selected.

A typical set of steps for development are summarized, and the errors which may enter the process at each stage are described. A generic triply-redundant and a generic distributed system are used as vehicles against which tools, techniques, and methodologies for verification are judged. Four levels of methodology depending upon the extent of automation are discussed.

The many new tools (computer programs which aid the practitioner) and techniques (procedures) are surveyed, and a discussion of how these help in the design and verification of the software for specific flight control functions is presented. A subjective evaluation of how these aids address the errors listed for each stage of development is offered. The tools and techniques are classified as dynamic or static. The first category refers to analysis of the execution of the code. These are aids in running and analyzing tests. Those in the second category examine the structure of the design and the code without executing it instruction by instruction. These tools cut development costs and promote quality software.

The technical correctness of the software may be shown by many approaches at each stage of development. Technical correctness is taken to mean that each flight control function performs as specified and that the integrated

structure, the data flows between functions, is correct. The validation of global system consistency and response to hardware errors is much more difficult.

The discussion of the four methodologies shows the several directions which may be used for incorporating tools in automating the design and its verification. The first methodology examines the considerations required for verification without machine aids. This establishes the detailed tasks. The second and third respectively add a set of tools and a software development system. The fourth considers the application of a completely formal methodology.

A very detailed review was made of the military standards that bear on digital flight controls, particularly MIL-F-9490D. Recommendations to modify the standards to include provisions for design and verification were made. These reflect the newer practices of software engineering.

SECTION I

INTRODUCTION

Analog autopilots were designed with special hardware to implement special control configurations. The control laws were synthesized by analysis and experiment on an analog computer which simulated the controls and the airframe. As the hardware was designed and built into testable units it was used in the simulation to replace its analog surrogate. This process of evolution continued to the "iron bird": A mock airplane representing the inertias and primary controls of the real thing. The mock control surfaces were driven by prototype servos, which in turn were controlled by a hardware assembly, its own components at various stages in the final design. Only the aerodynamic and dynamic reactions of the airplane remained on the analog computer. But this did not end the simulation and test sequence: before the first flight tests and after the autopilot was installed in the airplane, an analog computer was connected to the airplane. The computer simulated sensor signals and dynamic response expected of the airplane while the autopilot system operated. A flight test followed to verify the design data, the system behavior, and the overall performance. Since the reliability of electron tubes, magnetic amplifiers, and capacitor and resistor circuitry was low, the autopilot system was not permitted very much control over the airplane. Nonetheless, the testing, simulation, and review checks followed the control law and hardware design closely through all phases of the development. These checks were expensive.

The astonishing advances of microelectronic technology have changed much of this. We now have undreamed-of reliability and capability from the circuitry. The level of authority of the autopilot system has gone from only stability augmentation, added as a few degrees of surface travel by a series servo, to complete control of the airplane. The technology for a total fly-by-wire system is at hand. Confidence in the systems engineering must now be established. Testing, simulation, and analytic validation are still required for this task, perhaps at a much higher and more critical level than in the analog days.

The high levels of performance, functionality, criticality, and indeed the complexity now require that each subsystem or component must be more thoroughly verified before being integrated into the testing prototype of the vehicle. It is far too expensive to test for errors which, by a little discipline, can be found at earlier stages. This is particularly true of the software and it is also particularly true within the software development process.

The traditional approach for demonstrating the correctness of flight control software is by reviews and tests.[1] While this was adequate for the single-channel system for the JA-37 Viggen interceptor,[2,3] it does not provide the

---

[1] D.G. Bailey, "Airborne Software Structured Development," Report No. ED-DGM 4000-310 (A), Honeywell Avionics Division, Minneapolis, Minnesota, February 6, 1978.

[2] D.G. Bailey, and K. Folkesson, "Software Control Procedures for the JA-37 Digital Automatic Flight Control System," AIAA Guidance and Control Meeting, San Diego, California, pp. 122-129, August 1976.

[3] D.G. Bailey, and K. Folkesson, "JA-37 Digital Automatic Flight Control System (DAFCS) Self-Test Development," AGARD Report No. AG-224, Integrity of Flight Control Systems. Neuilly Sur Seine, France: North Atlantic Treaty Organization, April 1977.

necessary confidence as the flight control systems become more complicated and more critical to flight safety. Although the JA-37 has full authority over the control of the airplane, the pilot can intervene and fly with the primary control system. This is not the case with a fly-by-wire system. The integrity of the software and the system design must be shown beyond question. The validation procedures must prove that the system functions as intended.

The objectives of this study are 1) to examine the issues involved in showing the correctness of digital flight control software, 2) to review the new tools and techniques, and 3) to integrate these into methodologies to cover the entire development process. The study is intended to give a comprehensive and balanced picture of the problems and practices involved in providing high quality software. This detailed analysis provides the background for establishing the correctness of software operation and how it should be reflected in the Air Force Specifications for Flight Control Systems, particularly MIL-F-9490D. The necessity for further research projects will also be evident.

The methodologies must catch errors as soon as they occur in the process of developing the software. The economic necessity of this is clear: An error in design specification which is not uncovered until system test forces the software to go through the entire process again and is therefore costly. More important, methodologies which can monitor the development at each stage also provide more flexibility and certainty in making changes to the system during design and in the field. A large part of the life-cycle costs of embedded systems is incurred in modifications and operational changes required to adapt and update the operational capabilities as the system matures.

The methodologies must reduce the need for testing. Testing is expensive and should be used primarily to validate the performance of the system, not to find errors. It is also difficult to construct tests that are effective early in the development cycle. One usually must wait until executable code has been written. However, some functions such as the calculations of control laws are best checked by tests or simulations. The new tools of static analysis and aids for dynamic tests save developers much tedious labor and provide checks which likely would not be performed without the mechanical help.

The software for flight control systems is only modestly complicated. There are no complicated while-do loops, data structures are elementary, and the functions generally may be separated to yield uncomplicated control structures. The function by function verification of the software is not difficult. The interaction of the facilities for failure management and the assessment of system response to hardware errors are less certain. These questions will be examined closely in this report. We must get down to a very detailed level to determine what can be demonstrated rigorously and what remains in question. To quote N. Wirth,[4] "In programming, the devil hides in the details."

OVERVIEW

The discussion begins with a review of the development process for digital systems. The literature reports many variations but the basic stages are

---

[4] N. Wirth, Algorithms + Data Structures = Programs. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., p. xvi, 1976.

universal. We choose to follow the report by D. G. Bailey.[1] The stages of development are:

1) System requirements

2) Systems analysis and design

3) Software analysis and design

4) Coding and checkout testing

5) Software integration and integration testing

6) Software and qualification

7) Shipment and installation

8) Software maintenance

Our main concern is with stages 3, 4 and 5. A catalog of errors is compiled from a literature review, the experience of previous projects, and from a study of the inputs and tasks of each development stage. The chief criterion for a verification methodology is how well it addresses these errors. The terms verification and validation are defined as the links which demonstrate the correctness of the steps in the development process. This introductory section concludes with a review of the literature.

Having outlined the general process of development, we sketch two flight control systems in Section 2 which illustrate the software functions required of current and projected flight control systems. The first represents triply-redundant configurations typical of current designs. The second is a hypothetical distributed computer system from which we can project the problems in verification likely to arise in such assemblies. These generic systems

permit us to be specific in the discussion of errors and verification techniques for flight control software.

In Section 3 the static, analytic, and dynamic methods for verification are reviewed. These tools and techniques are rapidly developing and will play an increasing role in cutting software costs and in improving software quality. Our study will assess how these methods apply to current and future practices for digital flight controls.

The first three sections provide the background of the software development cycle, the typical flight control requirements, and the many tools and techniques to support the discussion of verification and validation methodologies in Section 4. We consider four levels: no machine aids, a methodology with an ideal set of tools, verification within an integrated development system (MUST), and the ultimate completely formal approach (SRI International's HDM). The comparison of the four levels provides a convenient format for the discussion. At this time we can only speculate on the utility of the last three levels; we may go through segments of the generic triply-redundant system by hand at our first level in order to obtain detailed analysis of the problems and results of verification. We then review the criteria for verification and consider how the methods cover the types of software errors listed in the next subsection. From these discussions we then review the certainty and confidence which may be placed in flight control software.

Section 5 considers providing for software verification in Air Force specifications. Section 6 summarizes the conclusions of this study. Section 7 lists topics for the future.

The current practice for the development of software for flight control systems is outlined in the next two subsections. While this is based on one particular approach, it presents a paradigm which will support our analysis.

SOFTWARE DEVELOPMENT AND ITS ERRORS

The software for flight controls, certainly in its initial stages, is developed in parallel with the design and construction of the prototype hardware for the system. This introduces complications and opens a major source of errors. Hence, we cannot jump to the development of the software but must also consider the systems aspects. In fact, an extended study would begin with the analysis of the requirement of the mission, a stage of system development which precedes our Stage 1: System Requirements. Errors in the scenarios and tasks of the mission will funnel down. For example, overlooking the windshear environment for landing may lead to an inadequate specification of stall margins on approach. Thus, inaccuracies in analyzing the mission requirements lead to errors in the inputs to the analysis of system requirements. The responsibility for the mission is with the Air Force, however a development methodology should check for omissions and inconsistencies and ask for clarification.

The details of the stages and the details of the tasks required at each stage are central to our analysis. In practice the stages overlap and steps are repeated because the requirements or the designs are changed. Our descriptions will consider each stage separately. The stages are described by their inputs, tasks, and outputs; the errors are cataloged as errors in

the inputs or errors in performing the tasks. At the coding stage, errors in the code are cataloged in considerable detail.

Stage 1: System Requirements

The objective of this stage is to prepare a preliminary system design that will be evaluated by the Air Force Program Manager. The documents produced are the technical and management proposals. While much attention is given to the system concept and configuration, the analysis of the software needs is limited to that necessary for planning and estimating costs. The inputs, tasks, and outputs of Stage 1 are summarized in Table 1.

The errors of this stage that are important to the subsequent software development are:

- Errors in Inputs--System requirements are:
  --Badly stated
  --Changed from those written
  --Incomplete or inconsistent
  --Overly restrictive
- Errors in Tasks
  --System requirements are misinterpreted and poorly documented
  --Configuration of the system is not correct
  --Size of the computer is inadequate
  --Hardware interfaces are unclear
  --Validation plan is not adequate

TABLE 1.

STAGE 1: SYSTEM REQUIREMENTS

Inputs	Tasks	Outputs
From the Program Manager: • Request for Proposal, including: --Statement of Work --Mission requirements --System requirements Functions Environments Reliability --Cost objectives Memory Throughput From Experience: • Specifications and reports on similar systems • Resources available to the project	• Analyze system requirements and characteristics, look for omissions and inconsistencies, refine and define system requirements. • Develop the configuration of the system • Estimate computer requirements and the computer loading • Prepare a development plan and cost estimates for the hardware • Prepare a development plan and cost estimates for the software • Determine requirements for supporting facilities and for software development tolls • Outline plan for system validation	• Definition of the system requirement • Definition of the configuration of the system • Proposals --Technical --Cost estimates --Management --Schedules • Memory and throughput estimates • Requirements for resources and facilities --Staffing --Software tools --Simulation facilities --Development laboratories • Validation plan

Software people commonly complain of poor descriptions of the hardware functions and interfaces. This problem recurs in subsequent development stages. The location of sensors in the aircraft can contribute to configuration errors if the locations are not accurately specified and compensation provided. In a triply-redundant system it becomes very difficult to allow for widely separated sensors because each computer program must then be individually treated. Inadequate documentation of the Air Force simulation facilities may make it difficult for the vendor to reproduce the simulation and perform the needed requirements analysis.

Stage 2: System Analysis and Design

In this stage the system configuration is refined and completed so that the hardware and software requirements can be carefully specified in a detailed document called the System Development Specifications, Part I. The specifications for the software requirements may be split into those for the operating functional software, those for the software necessary for the acceptance tests of the hardware, and those for the specialized software for the built-in-test (BIT) functions. Our discussion will center on the functional software. Table 2 shows the Stage 2 inputs, tasks, and outputs.

The errors in Stage 2 are:

- Errors in Inputs--The system requirements are:
  --Misinterpreted
  --Poorly documented
  --Data defining the system is incomplete
  --Descriptions of hardware and peripheral equipment are inaccurate or incomplete

TABLE 2.

STAGE 2: SYSTEM ANALYSIS AND DESIGN

Input	Tasks	Output
System requirements System configuration User's environment • Constraints • Interfaces • Operating procedures Description of design and operation of computer and peripheral equipment Description of machine language and command structure Description of special processing and storage requirements Description of available tools Documents from similar projects	Establish the system configuration • Review the preliminary configuration • Perform additional analysis and simulations • Conduct hardware/software trade-off Prepare the system development specifications (DS Part I) Update and document software development plans Review programming language selection Update the cost estimates Conduct software concept review Define software test plans Define software configuration management plans, procedures Gather documentation for support facilities, software, and manuals for the target computer and peripheral equipment	Software requirements specification (DS Part I) Programming language selection Software development plan, updated Master test plan Configuration management plan, software change procedures Revised estimates of costs and staffing needs Requirements for support facilities and software development tools

- Errors in Tasks
  - --System configuration is wrong because of poor hardware/ software trades, awkward executive control structures, faulty computer synchronization schemes
  - --Documentation of the specifications is incomplete
  - --Poor selection of the computer language
  - --Software test procedures are poorly designed
  - --Concept review is badly done

Since there may be a transfer from systems people on Stage 2 to software people on Stage 3, the documentation of the software requirements is very important. Further, errors or omissions in the specifications do not merely filter down to the following stages but tend to fan out and be compounded. A study of 11 projects showed that the majority of the software errors evolved from faulty specifications.[5] The specifications may be inaccurate because certain data, for example the control laws, are not correctly given.

Stage 3: Software Analysis and Design

We now are in the realm of software. A design is prepared to meet the requirement specifications defined in the previous stage. The description of the design is part of the preliminary product specification, also called the detailed design specification (DS Part II). The details of the testing procedures are now filled in: Test data are selected; the supporting software tools and facilities are limbered up; a preliminary design review (design walk-through)

[5] B.W. Boehm; R.K. McClean; and D.B. Urfrig, "Some Experience with Automated Aids to the Design of Large-Scale Reliable Software," Proceedings of the International Conference on Reliable Software, Los Angeles, California, pp. 105-113, 21-23 April 1975.

is conducted to verify that the design complies with the requirements. Table 3 shows the Stage 3 inputs, tasks, and outputs.

The errors in this stage are:

- Errors in Inputs
  - --Lack of coordination between Stage 2 and Stage 3
  - --Poor documentation on the hardware
  - --Inadequate data on the configuration
- Errors in Tasks
  - --Functions are partitioned incorrectly
  - --Software organization is insecure
  - --Poor use of software standards, guidelines, and programming methodology
  - --Incorrect algorithms
  - --Documentation is inadequate and incomplete

The documentation input for Stage 3 may now become confused because there are customer specifications, the DS Part I split into two or three pieces, and there may be parts of a preliminary version of DS Part II. Changes introduced by the customer at this point add to the confusion. The partitioning of functions into modules may call for circular references between modules. This makes checkout and subsequent changes very difficult. If modules are made too large, complete testing is difficult and errors in interfaces, which are not caught until integration testing, increase. The new design methodologies may be abused. Gerhart and Yelowitz[6] provide

[6] Gerhart, S., and Yelowitz, L., "Observation of Fallibility in Applications of Modern Programming Methodologies," IEEE Transactions on Software Engineering, Vol. SE-2, No. 3, September 1976.

TABLE 3.

STAGE 3. SOFTWARE ANALYSIS AND DESIGN

Input	Tasks	Output
Software requirements specification (DS Part I) System development specification (DS Part II) Documentation on computer hardware and peripheral equipment Description of support software	Partition software functions into modules and submodules to form a hierarchy Define module interfaces Package the modules into an executive structure. Define the timing constraints. Define data structures, storage allocations, formats, conversions Define programming standards, guidelines Derive special algorithms Update computer load estimates Prepare preliminary systems documentation and DS Part II Develop software module and integration test procedures, software qualification tests, post installation tests, BIT verification procedures Check out software tools and facilities Produce documentation for preliminary design review and conduct the review	Detailed design specifications (DS Part II) Software qualification test procedures Post-installation test procedures BIT verification procedures Preliminary design review proceedings

some examples. The problems in keeping the documentation current and consistent are generally underestimated.

Stage 4: Coding and Checkout Testing

Producing the code is the most visible and least interesting stage of development. It amounts to only about 20 percent of the software cost. If the detailed design specifications have been carefully documented and reviewed, this stage is routine. The inputs, tasks, and outputs are listed in Table 4.

The errors in this stage are the easiest to catalog and have been extensively studied. The TRW report[7] gives an excellent discussion. A summary of this work is reported in Reference 8. As with the previous stages, we offer a general list of errors but add a detailed list of categories selected from Reference 7.

- Errors in Inputs
  --Incomplete or erroneous design specifications
  --Poor documentation of the hardware
- Errors in Tasks
  --Coding errors:
  Misinterpretation of language constructs hardware

---

[7] T.A. Thayer, et al., "Software Reliability Study," Final Technical Report No. RADC-TR-76-238, TRW Defense and Space Systems Group, Redondo Beach, California, August 1976.

[8] D.K. Lloyd, and M. Lipow, Reliability: Management, Methods and Mathematics. Redondo Beach, California: Second edition, published by the authors, 1976.

TABLE 4.

STAGE 4: CODING AND CHECKOUT TESTING

Input	Tasks	Output
Detailed design specification (DS Part II) Computer and peripheral equipment descriptions Support facility and software descriptions Module test plans	Code program modules Keypunch source statements Establish plans for checkout testing of program modules Debug program modules Checkout module execution on simulation or prototype system Assemble or compile complete program for the integration checkout phase	Source listings for each program module Object code for each program module Updated system documentation Updated software component for DS Part II Source listing for entire program

Operations overflow

Scaling and approximation errors

Self-modification of instructions

Sequencing and branching errors

Incorrect loop structures

Loss of index or state data

Mishandling of singular and critical values

Incorrect values for constants

Unreachable code

Uninitialized variables

--Code not reviewed

--Typographical errors

--Incomplete testing of every module

--Incorrect implementation of design specifications

--Poor documentation

On a large project, Boehm[9] found that 28 percent of all software errors were clerical or typographical. Code produced on developmental projects often has many abandoned segments which were left in under the supposition that they would be removed if the project went into a product stage. In one project, erroneous functions were bypassed so the project could proceed to testing the integrated code only to find that some of these bypassed functions were forgotten until the lack of functionality was discovered in the last phases of testing. Thus, isolated code may indicate more severe problems than wasted memory space. Often code is not reviewed before it is subjected to testing. This obviously leads to a high error count.

---

[9] B.W. Boehm, "Software and Its Impact: A Quantitative Assessment," Datamation, pp. 48-59, May 1973.

Some code is better than others. Much has been written about quality and the characteristics which prompt one to call a specific code good. Boehm, et al.[10] define a long list of quality attributes. Code that is well-organized is generally easy to read, easy to modify, easy to test, and so on. It may not be the most efficient in the number of instructions.

The extent to which coding errors may be categorized is illustrated by this partial list:[7]

Computational Errors/Faults

- Incorrect operand in equation
- Incorrect use of parentheses
- Sign convention error
- Units or data conversion error
- Computation produces an over/under flow
- Incorrect/inaccurate equation used
- Precision loss due to mixed mode
- Missing computation
- Rounding or truncation error

Logic Errors/Faults

- Incorrect operand in logical expression

[10] B.W. Boehm, J.R. Brown, M. Lipow, "Quantitative Evaluation of Software Quality," Proceedings of the Second International Conference on Software Engineering, San Francisco, California, pp. 592-605, 13-15 October, 1976.

- Logic activities out of sequence
- Wrong variable being checked
- Missing logic or condition tests
- Too many/few statements in loop
- Loop iterated incorrect number of times (including endless loop)
- Duplicate logic

Data Handling Errors/Faults

- Data initialization not done
- Data initialization done improperly
- Variable used as a flag or index not set properly
- Variable referred to by the wrong name
- Bit manipulation done incorrectly
- Incorrect variable type
- Data packing/unpacking error

Interface Errors/Faults

- Wrong subroutine called
- Call to subroutine not made or made in wrong place
- Subroutine arguments not consistent in type, units, order, etc.
- Subroutine called is nonexistent

Data Definition Errors/Faults

- Data not properly defined/dimensioned
- Data referenced out of bounds
- Data being referenced at incorrect location
- Data pointers not incremented properly

One of the purposes for compiling these lists was to determine how well the tools and techniques for verification covered a particular error category. We are also following that approach.

The four stages of development that have been discussed are the origins of the errors addressed by verification and validation analysis. Outlines of the next four stages are included to complete the picture.

Stage 5: Software Integration and Integration Testing

This is the final stage before the software is complete and is then released. The documentation is updated and a final critical review of the design is made. The software specification, DS Part II, describes the software and helps in any field modifications and other changes. However, programmers say that the only genuine specification is the assembly listing itself. The integration of the modules and the subsequent testing must provide confidence that the software is error-free and is ready for qualification testing. Table 5 provides a summary of the inputs, tasks, and outputs.

TABLE 5.

STAGE 5: SOFTWARE INTEGRATION AND INTEGRATION PROGRAM

Input	Tasks	Output
Program module source and object listings **Detailed specifications of each module** **Checkout procedures for each module** **Integration test plan**	Run diagnostic tests on computer and peripheral equipment Validate deliverable support software Integrate modules into the operational code Check out total hardware/software operation Analyze test results Initiate modifications to program modules and recheck results Document test results Update, maintain documentation Begin program to train customer and plan for post-delivery maintenance Conduct the critical design review of the software	Operational software object code Software Specifications DS Part II System documentation Test reports

Stage 6: Software Qualification

The tests at this stage must demonstrate that the software complies with the requirements of DS Part I. It is the first step in validating total system performance as required by the Air Force. Table 6 summarizes the inputs, tasks, and outputs. The hardware on which the tests are run must already have been qualified and procedures must be instituted to control hardware and software changes.

Stage 7: Shipment and Installation

The summary is provided in Table 7. The second step in validation is done at an Air Force facility to demonstrate that the complete system complies with the functional and performance requirements defined for the system.

Stage 8: Software Maintenance

The last stage in our scenario is ultimately for many systems the most expensive. Some reports attribute more than 50 percent of the life-cycle costs of software to maintenance. These costs are not in correcting software errors but in making the changes necessary to improve the system and to adapt the system to new operational requirements. The software is the flexible part. Unfortunately, some of the changes are needed to fix deficiencies in the design. The inputs, tasks, and outputs are listed in Table 8.

VERIFICATION AND VALIDATION

Having sketched out a sequence of development stages and having listed the errors which may occur, we can review the process currently used to correct

TABLE 6.

STAGE 6: SOFTWARE QUALIFICATION

Inputs	Tasks	Outputs
Design specification, Part II System documentation • Schematics • Input/output lists • Etc. Object code Software qualifcation procedures	Review the results of the integration tests and review the qualification procedures Conduct qualification tests Complete management procedures for controlling changes Obtain Program Manager's approval *of test results* Conduct audit to make certain that the software is ready to ship to the Program Office	Qualifications test report Engineering change order procedures

TABLE 7.

STAGE 7: SHIPMENT AND INSTALLATION

Inputs	Tasks	Outputs
Design specification, Part II Object tapes or decks Test reports Guides and manuals Specifications for producing the software Post-installation test plans	Install and validate the system performance Conduct post-installation tests Conduct demonstration tests	Acceptance by the Air Force

TABLE 8.

STAGE 8: SOFTWARE MAINTENANCE

Inputs	Tasks	Outputs
Software documentation Software code Test procedures Change control procedures	Develop a plan for software maintenance Review change control procedures for field use Define requalification procedures	Revised software documentation Revised software Software maintenance plan, change procedures and retesting plans

the errors. The development process is shown in Figure 1. Verification may be defined as the demonstration that each stage is correct and that the software program is a correct rendition of the software design. Verification asserts the integrity of the process, step by step. Validation may be defined as the demonstration of the correct performance of the entire package. Validation shows that the system performs the functions listed in the system requirements and does not perform any unintended functions.

The verification of the early system stages is done by analysis and reviews. The verification of software uses tests in addition to the analysis and reviews. Here, the new tools and techniques can add confidence and cut the costs of the process. However, attention to organized methods and documentation greatly improves the earlier verifications. Validation relies on testing the actual flight hardware and software in a simulated environment to prove that the system functions as required.

Testing of the software occurs at four stages in the development cycle. At each subsequent stage the scope of testing and configuration control is increased. A thorough review of the procedures is required to ensure that the testing is sufficient. Review by the Air Force further ensures that these procedures are complete.

Module Testing and Program Verification

The first stage of testing is performed to verify the functionality of each module as it is coded. Then the modules are integrated into the total program. The program is then tested to show that it runs as it was designed to.

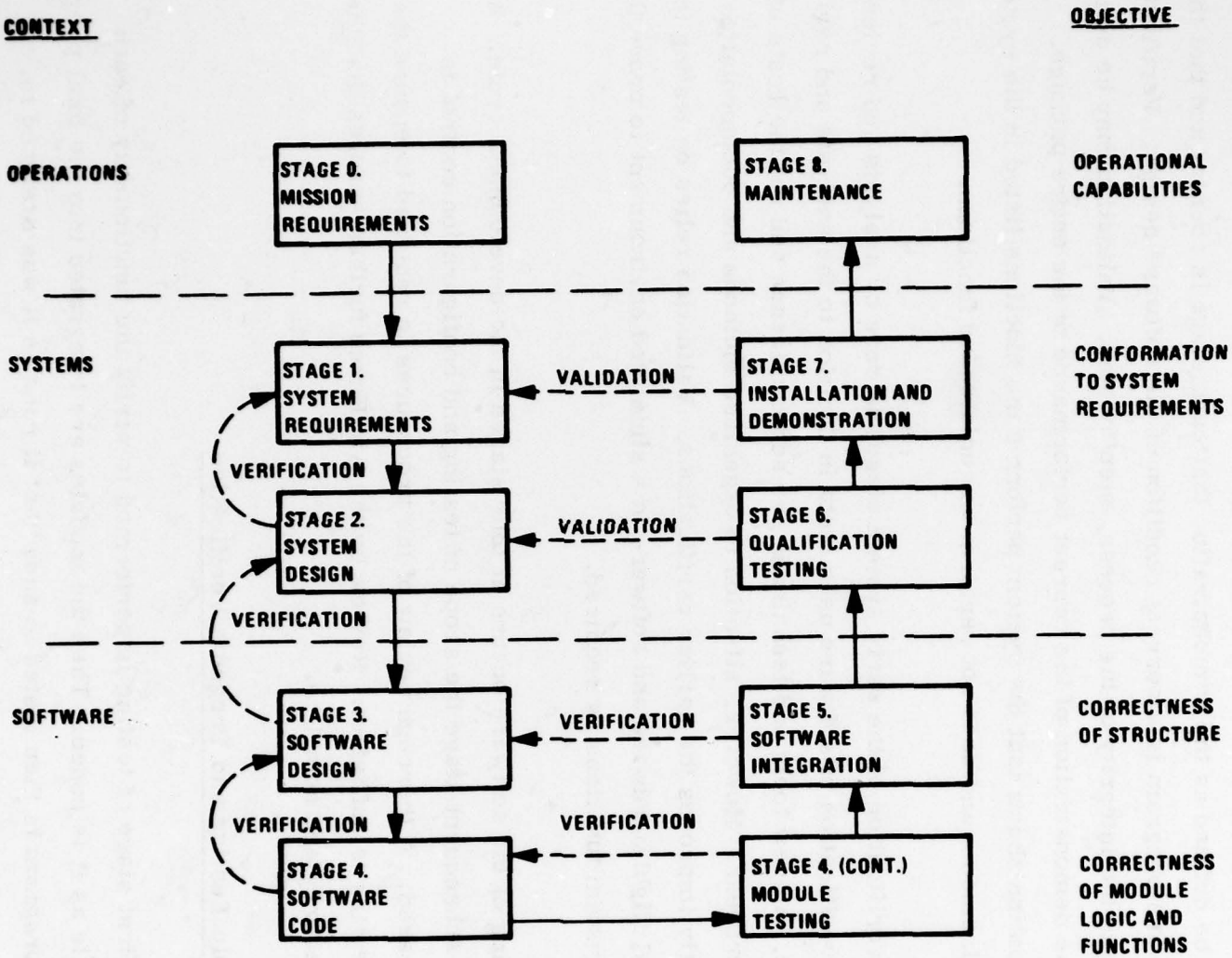


Figure 1. Development Process

Program Validation

The second stage of testing is end-to-end validation of the complete software operating on the flight system. This testing is done in a simulated environment using the test equipment. It verifies each requirement of the software and systems specifications. Those requirements verified previously at the module level need not be re-verified at this stage.

System Validation by the Air Force

The third level of validation testing is performed on the iron bird simulation of the airplane. This testing integrates the proposed design with all other systems in the airplane and is a complete functional test of the systems hardware and software.

Flight Testing

A fourth level of validation occurs during flight tests in the airplane. A particular set of tests used in the iron bird simulation is repeated during flight testing.

The Re-Verification Process

Procedures must be established to show that changes made to the software are correct. If a change in requirements is ordered after the software has been delivered, the entire development process must be repeated. If an error in the design of a module is detected during testing, then:

- Redline (edit) the documentation of the module design
- Redline the test procedure for the module
- Code the required changes
- Walk through the design and code changes
- Test the modified module
- Insert the module into the system program
- Continue the verification testing with the revised program

To determine the retesting which is necessary,

- List modules and data locations which are changed
- List modules which access the altered code or data
- Define tests to show that all modified code performs correctly

The process must ensure that the changed software is verified as completely as the original software.

REVIEW OF THE LITERATURE

Since testing and the supporting analysis may add up to over half the cost of a software project, there is much research activity in this area to lower the cost and provide reliable service. The most significant improvement has been the use of careful and deliberate program structures and documen-

tation of program designs. This trend has reached the textbook stage.[11-14] The text by Schneider, et al.[13] is a sample of the new approach to teaching programming; Reference 14 shows how to construct formal proofs along with the development. The adoption of company guidelines[1,15] has stimulated the use of more organized software development than in the past.

Reifer and Trattner[16] give a long and useful list of the tools and techniques we are examining. Other lists are provided by Benenati and van Dam[17] and by Miller.[18] General surveys of V&V techniques are reported by Schlicht,

---

[11] R. C. Tausworthe, Standardized Development of Computer Software. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1977.

[12] E. Yourdon, Techniques of Program Structure and Design. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1977.

[13] G. M. Schneider, S. W. Weingart; and D. M. Perlman, An Introduction to Programming and Problem Solving with Pascal. New York: John Wiley & Sons, 1978.

[14] S. Alagic, and M. A. Arbib, The Design of Well-Structured and Correct Programs. New York: Springer-Verlag, 1978.

[15] D. Lane, "Structured Programming Guidelines," Report 776-12882, Vol. I and II, Honeywell Avionics Division, St. Petersburg, Florida, September 1977.

[16] D. J. Reifer, and S. Trattner, "A Glossary of Software Tools and Techniques," Computer, Vol. 10, No. 7, pp. 52-61, July 1977.

[17] C. J. Benenati, and A. van Dam, "An Informal Survey of Software Engineering Tools and Methodologies," Raytheon Submarine Signal Division, Providence, Rhode Island, December 7, 1977.

[18] E. F. Miller, "Program Testing Tools: A Survey," presented at MIDCON, Chicago, Illinois, November 8, 1977.

et al.[19] and Smith.[20] A very good review may be found in Section IX of the Hughes Report.[21] This is somewhat parallel to the work by Thayer, et al.[7] Section 4 of that report is one of the foundations of our study. For reviews of current V&V developments the paper by Ramamoorthy and Ho[22] is good and the paper by DeWolf and Wexler[23] is excellent. The paper by Hartwick[24] presents an analysis of errors and how they are detected by specific methods, an approach that we are following. There are an incredible number of papers in the periodicals. We found those by Fairley,[25] Huang,[26] Panzel and

---

[19] R. A. Schlicht; W. M. Brueggeman; and G. W. Staley; "1975 ADG Software Program: Verification/Validation," Vol. I-V, Honeywell Systems and Research Center, Minneapolis, Minnesota, 1975.

[20] R. L. Smith, "Validation and Verification Study," from Structured Programming Series, Vol. XV. Gaithersburg, Maryland: International Business Machine Corporation, May 22, 1975.

[21] R. N. Furtaw; D. T. Edson; R. P. Teutsch; and J. S. Steiner; "Aircraft Avionics Tradeoff Study," Report No. ASD/XR 73-19, Hughes Aircraft Co. Canoga Park, California, pp. 307-369, November 1973.

[22] C. V. Ramamoorthy, and S. F. Ho, "Testing Large Software with Automated Software Evaluation Systems," IEEE Transactions on Software Engineering, Vol. SE-1, No. 1, March 1975, pp. 46-58. Also in R. T. Yeh (ed.), Current Trends in Programming Methodology, Vol. 2. Englewood Cliff, New Jersey: Prentice-Hall, Inc., pp. 112-150, 1977.

[23] J. B. DeWolf, and J. Wexler, "Approaches to Software Verification with Emphasis on Real-Time Applications," AIAA Computers in Aerospace Conference, Los Angeles, California, pp. 41-51, October 31 to November 2, 1977.

[24] R. D. Hartwick, "Test Planning," AFIPS Conference Proceedings, Vol. 46, pp. 285-294, 1977 National Computer Conference.

[25] R. E. Fairley, "Tutorial: Static Analysis and Dynamic Testing of Computer Software," Computer, Vol. 11, No. 4, pp. 14-23, April 1978.

[26] J. C. Huang, "Program Instrumentation and Software Testing," Computer, Vol. 11, No. 4, pp. 25-32, April 1978.

Darringer,[27] and King[28] in the April 1978 issue of Computer to be particularly useful for our study.

Automatic methods of verifying programs are under study.[29-31] Of special importance to this study is the work of Maurer[32] on proving programs in assembly language and the work by SRI International.[33] SRI is attempting to construct an entire formal methodology. Honeywell made a preliminary investigation of these techniques for flight control software.[34]

---

[27] D.J. Panzl, "Automatic Software Test Drivers," Computer, Vol. 11, No. 4, pp. 44-50, April 1978.

[28] J.A. Darringer, and J.C. King, "Applications of Symbolic Execution to Program Testing," Computer, Vol. 11, No. 4, pp. 51-60, April 1978.

[29] B. Elspas; K.N. Levitt; R.J. Waldinger; and A. Waksman; "An Assessment of Techniques for Proving Program Correctness," ACM Computing Surveys, Vol. 4, No. 2, June 1972.

[30] S.L. Hantler, and J.C. King, "An Introduction to Proving the Correctness of Programs," ACM Computing Surveys, Vol. 8, No. 3, pp. 331-353, September 1976.

[31] R.L. London, "Perspectives on Program Verification," in R.T. Yeh (ed.), Current Trends in Programming Methodology, Vol. 2. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., pp. 151-172, 1977.

[32] W.D. Maurer, "Proving the Correctness of a Flight-Director Program for an Airbourne Minicomputer," SIGMINI/SIGPLAN Meeting Proceedings and SIGPLAN Notices, Vol. 11, No. 4, pp. 103-108, April 1976.

[33] L. Robinson, and K.N. Levitt, "Proof Techniques for Hierarchically Structured Programs," in R.T. Yeh (ed.), Current Trends in Programming Methodology, Vol. 2. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., pp. 173-196, 1977.

[34] W.E. Boebert; J.M. Kamrad; and E.R. Rang; "The Analytic Validation of Flight Software: A Case Study," NAECON 1978, Vol. 1, pp. 242-248, May 26, 1978.

The experience in developing the software for the first digital flight control system to go into production is recorded in References 2 and 3. References 35 and 36 discuss methods of demonstrating the synchronization of redundant computers. Besides the development of special tools, there are significant developments in systems of tools to do a comprehensive job of software production and verification.[37,38]

The last paper we cite in this general summary is the widely quoted paper by Goodenough and Gerhart.[39] The authors suggest using decision tables for analyzing test cases. This works well for setting up tests for the logic and redundancy management modules of flight control systems. They also begin a theory of test data selection, which we have not found useful.

---

[35] J. L. Peterson, "Petri Nets," ACM Computing Surveys, Vol. 9, No. 3, pp. 223-252, September 1977.

[36] A. F. Babich, "Proving the Correctness of Parallel Programs," IEEE Computer Society, Paper R-78-21, February 1978.

[37] S. H. Saib; J. P. Benson; and R. A. Melton; "Software Quality Laboratory," presented at AIAA Computers in Aerospace Conference, Los Angeles, California, October 31 to November 2, 1977.

[38] T. A. Straeter; E. C. Foudriat; and R. W. Will, "MUST: An Integrated System of Support Tools for Research Flight Software Engineering," Paper no. 77-1459, AIAA Computers in Aerospace Conference, Los Angeles, California, pp. 442-446, October 31 to November 2, 1977.

[39] J. B. Goodenough, and S. L. Gerhart, "Toward a Theory of Test Data Selection," IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, pp. 156-173, June 1975. (Also revised in R. T. Yeh (ed.), Current Trends in Programming Methodology, Vol. 2. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., pp. 44-79, 1977.)

SECTION II

GENERIC FLIGHT CONTROL SYSTEMS

Two hypothetical systems are outlined against which the verification tools, techniques, and methodologies may be weighed. The first is a triply-redundant system representing current practice. It is a generalization of an experimental system which was carried as far as a total simulation. Its documentation was the assembly code and flow charts. For this study, the code was interpreted in HIPO charts (hierarchy plus input-process-output charts). Much more will be said about this later. The triply-redundant system is not the most elegant design but it illustrates the functions that must be provided at the level of detail necessary to discuss the problems of verification. The second system is distributed. It is much more speculative. We will attempt to show that if the computations are distributed according to function, no new types of problems for verification arise. However, more complicated configurations are possible in which the executive control is very fluid. This makes the demonstration of the failure management mechanisms very difficult.

TRIPLY-REDUNDANT CONTROLS

The generic system chosen uses three computers and four sets of sensors. The key features for this architecture are:

- Two signal select nodes: one at the sensor inputs, the other at the servoactuator outputs.

- The system tolerates first and second failures and is failsafe for third failures.
- Macro synchronization or frame synchronization is achieved with external hardware.
- Micro synchronization or within-frame synchronization is achieved with software in conjunction with the real-time clocks.
- The watchdog timer detects gross computer failures that cause loss of real-time control.
- Data exchanges between channels are initiated by the sender.
- Comparisons of the outputs are performed by software.
- Built-in-tests are run in conjunction with the control laws to provide confidence during single computer operations.

The Configuration

The combination of the three computers and the four sets of sensors is illustrated in Figure 2. Each computer reads the signals from its own "local" sensor group; it also reads the signals from the standby sensor group. The local signals are exchanged between the computers. This allows cross-channel monitoring of the sensor inputs. The second point of comparison is performed before sending out the command signals. The calculated outputs are interchanged and the final outputs are selected after the comparisons have been made. For this study we assume that the system is connected in a simulation so the output signals are summed to drive a

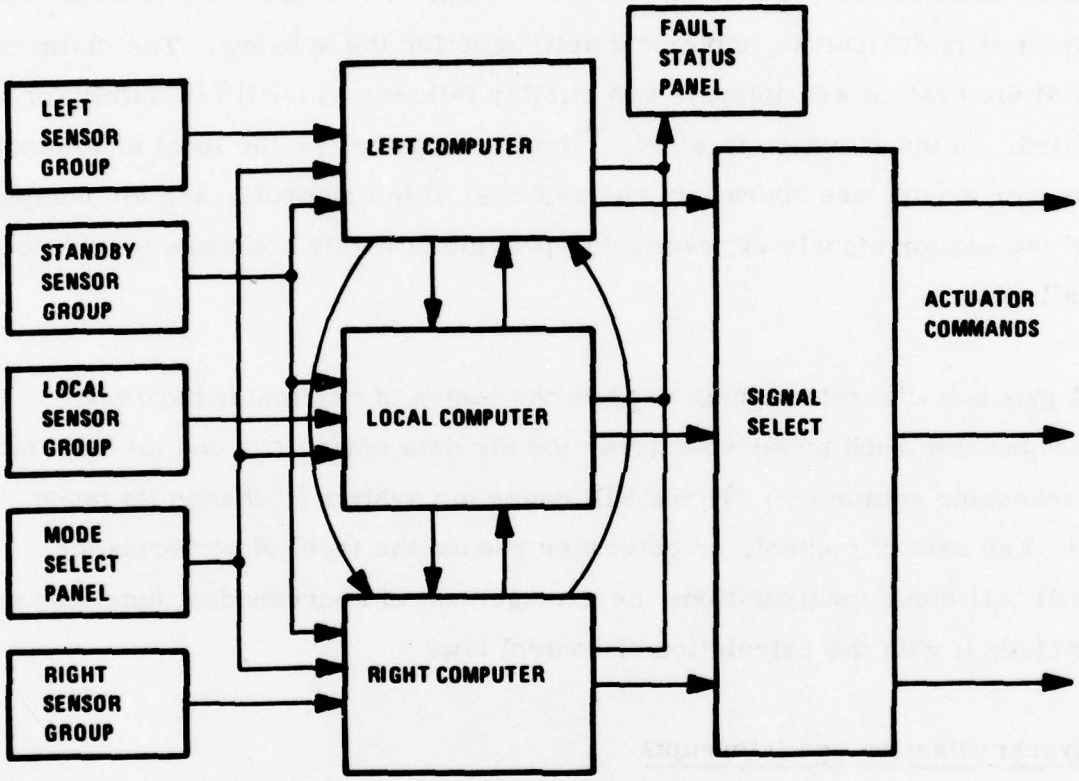


Figure 2. The Configuration

single simulated actuator. Appropriate changes must be made in the selection mechanization for an operational configuration. A single set of mode select signals, servo status signals, and discrete inputs such as landing gear status signals is sent to each computer.

Each channel performs in-line or self-tests. The fourth sensor is provided since it is difficult to implement self-tests for the sensors. The claim is that the system will tolerate two similar failures and will fail safely for the third. In the situation in which a single computer and the local and standby sensor groups are operating, the self-test of the computer and the comparison of the sensor signals is assumed to provide fail-safe response to additional failures.

A group of discrete signals reports the status of particular hardware components such as servo valves, the air data computer, and the inertial measuring equipment. These will cause the system to change its mode, shed an axis of control, or otherwise reduce the level of performance. We will call these manipulations the management of nonredundant functions and include it with the calculation of control laws.

Synchronization and Interrupts

The operation of the computers must be synchronized with sufficient accuracy to allow the exchange of data. A halt-release synchronization using special external hardware to halt the faster computer until the slower one catches up is used for synchronizing each frame and for synchronization on start-up. Synchronization to a local external real-time clock is used for data transfers. Experience shows that the three real-time clocks are sufficiently precise for this purpose. Additional details are provided in Appendix A.

A momentary power interrupt will cause the computer hardware to reset the program counter at the beginning of the program. This is the only interrupt that is provided.

Rate Structure

The simple executive rate structure of the program is diagrammed in Figure 3. The functions which must be calculated rapidly, such as the inner loop laws, are in the fanned-in part of the structure. These are computed in alternate frames or cycles through the rate tree. The slow functions, such as the control gains, are in the fanned-out part. These are each computed only every eighth frame.

The computer is halted by a software command when the program reaches the end of the loop. It is released by a signal from the external synchronizing hardware after the slowest computer reaches this point in its program.

Flight Control Functions

Each of the functions listed in Figure 3 is now briefly described and the software requirements are analyzed.

Initialize Computer--The program starts here when the power is switched on or when a brief interrupt of power has occurred. The scratchpad memory is initialized if neither right nor left computers are operating and the discrete input power monitor is set. This function is not complicated. Its design and functioning can be explained adequately by flow chart or HIPO for design verification. The subroutine which sets the initial values may supply the

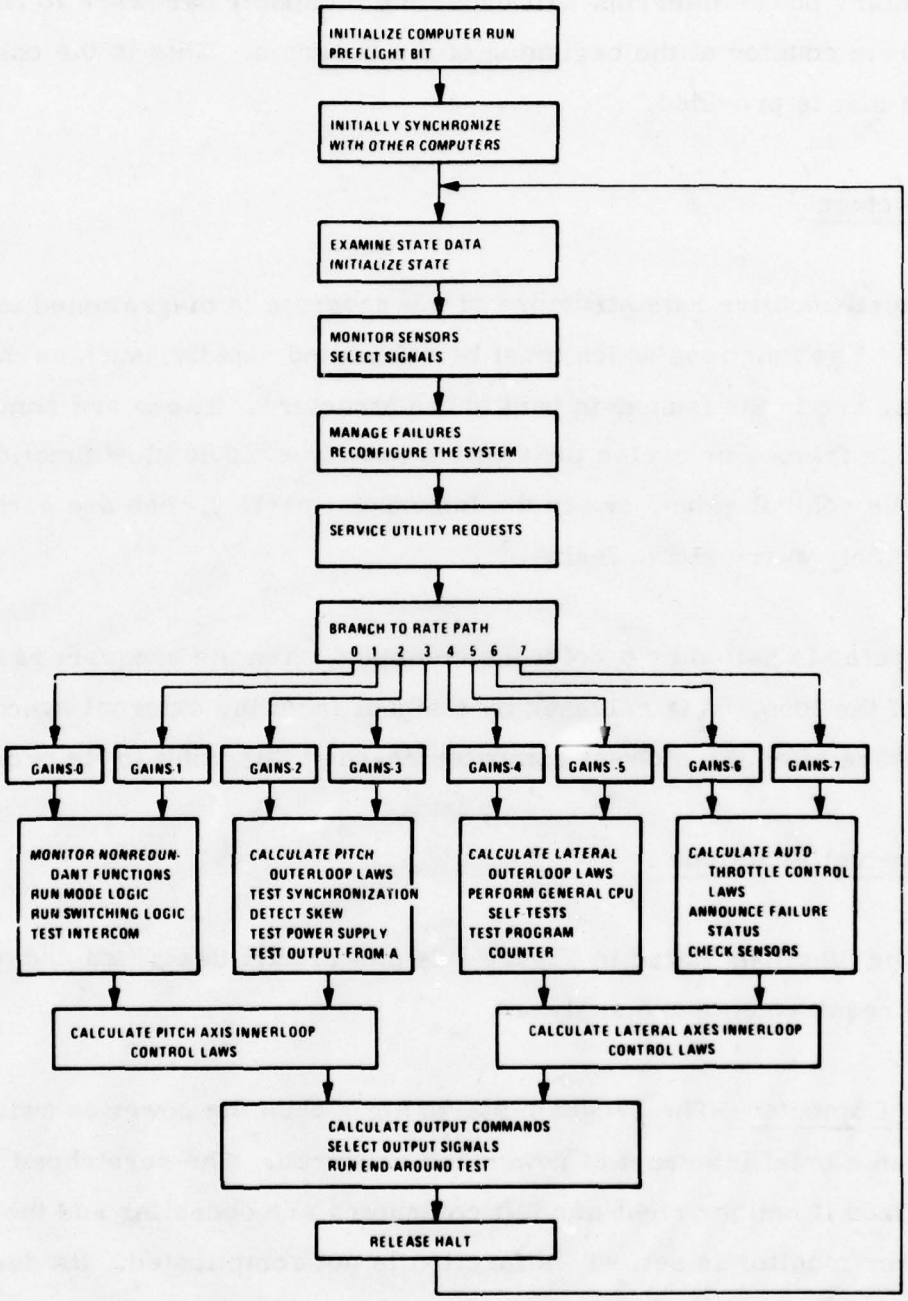


Figure 3. The Rate Structure

wrong value (a change was made in the program module but was not followed through in the initialization) or it may omit a value.

Run Preflight Built-In-Test (BIT)--If neither right nor left computers are running and the power monitor discrete is not set, the preflight BIT is called. A counter is used to allow several failure indications before the computer is shut down. The design of this program may be flight-critical since practically and theoretically it must detect failures in preflight to ensure that the reliability design calculations are valid during flight.

Initially Synchronize with Other Computers--The signals from the external synchronizing hardware are used to provide the information that causes synchronization on start-up. The synchronization must also determine the proper path in the rate structure. The module is rather complicated but the synchronization may be described using Petri nets. The process may be completely verified by simulations or tests since there are not an unmanageable number of possibilities. It is certain that the synchronizing hardware and software of this system may be greatly simplified and improved. Details are provided in Appendix A.

Exchange State Data, Initialize State--The data that describe the filter states, the autopilot mode, and the discrete signals that are packed into the executive words are exchanged between the computers. The order of synchronization must be analyzed to determine which computers need the initialization. A common error is the omission of a particular state. In the generic system, the hardware requires a within-frame synchronization to effect data transfer. Newer computers allow autonomous input/output (not under software control) so the tedious "path balancing" in the software

is not required. The module and its logic to analyze the order of the initial synchronization of the computers is easy to describe.

Monitor Sensors, Select Signals--Each computer reads its own set of sensors and the standby set of sensors. The local sensor signals are then exchanged and compared. Depending on which combination of computers is operating, four selection policies are provided. Counters are used to reduce nuisance failure indications. A failure flag system shows the status of the sensors and latches the selection policy accordingly. This is a large and very complicated program component. It is certainly critical to flight safety. It is possible to design this component using finite state automata to get an adequate description.

Manage Failures, Reconfigure System--The fault status is analyzed and appropriate action is taken. The output channels are shut down if the corresponding signal shows failure of the servo. Again, depending on which combination of computers is operating, one of four program segments is followed. The consistency of the failure indications between the computers is analyzed. The shut-down sequence is provided here. The organization of this program component is difficult to describe; it is critical to flight safety.

A higher level of redundancy may be achieved for some situations without much added processing. There are combinations of three failures (while three channels are operating) and combinations of two failures (while two channels are operating) which are considered to be fail-safe. If a pair of sensors can support a computer, the comparison of sensors and the self-tests of the computer provide the safety in the event of another failure.

However, certain combinations of three failures do not leave the system fail-safe because the valid sensor signals cannot be accessed through a failed computer. This results from our configuration in which each computer can access the standby sensor and its local sensor but must rely on the left or right computer to transmit the signal from the left or right sensor. Tests for these combinations are included; if detected, the system is shut down.

Service Utility Requests, Branch to Rate Path--These are elementary functions. The design and verification do not present any problems. A provision for refusing any utility requests in flight must be added.

Calculate Gains--Most autopilots adjust the control loop gains according to schedules which depend on air-data parameters such as altitude, Mach number, airspeed, and angle of attack. These parameters of the flight envelope change slowly. Hence, the gains need not be recomputed rapidly and are done in the slow part of the rate structure. The gain schedules are piecewise-linear functions and thus require a number of parameters for their description. Errors in data or mechanization are hard to detect. The redundancy requirements are not clearly defined in our generic system. The calculation of gains is usually checked by simulations.

Monitor Nonredundant Functions--There are a number of non-flight-critical functions which when failure is detected cause a shedding of capability. The analysis and design of these functions is not complicated.

Run Mode Logic--The control panel inputs and dynamic state are analyzed to produce the required mode configuration and to set flags that determine the course of the control law computations. These functions may be designed

and described by Boolean logic. Verification of the design is thus mathematical and it can be made completely convincing with some attention to details.

Run Switching Logic--Additional changes in the control law computations are made because of the dynamic state. These have only momentary effects and are separated from the mode logic to give a better modularization of the computations. These are also easy to handle precisely.

Test Intercom--The transmission of data between computers is tested by exchanging sets of stored test words. Counters are used to avoid nuisance fault indications. The test and its implementation are easy to describe.

Calculate Pitch Outer Loop Laws--The outer loop control laws require relatively slow calculations. Inputs are from the air data computer, IME or the control panel, pitch altitude, pitch rate, normal acceleration, etc. The modes may be altitude hold, altitude capture, constant rate-of-climb, etc. Computations that require some care are those which use integration to synchronize upon mode engagement so that there are no switching transients or bumps.

The calculation of the control laws may be easily and completely verified by determining the frequency response on a simulation. An overflow of these numerical calculations is the major concern. The difficulties are in describing the mechanisms for controlling the modes of calculations. The traditional analog diagrams are not convincing.

Calculate Lateral Outer Loop Laws--Modes such as heading hold, VOR track, and heading capture are controlled by lateral axes commands.

Calculate Auto Throttle Control Laws--The throttle control is not flight critical and is not made redundant. Monitoring and self-tests are used to revert to manual control on failures.

Test Synchronization--Test words are transmitted immediately after the halt-release. The procedure depends on the computer hardware and has not been brought up to date.

Detect Skew--The time skew between channels is checked by exchanging a data word and comparing it to the value expected.

Test Power Supply--Eight voltages from the power supply are read in by the analog/digital converter and compared to the expected values with an assigned tolerance.

Test Output From A--An output fault is declared and the system is disengaged if the digital/analog converter reports not ready or the error between the signal sent out and its rendition read back exceeds a given tolerance. The verification of this test is used as an example in Appendix C.

Perform General CPU Self-Tests--The processor self-test, the parity checker test, memory sum check, memory addressing test, watchdog check, scratchpad sum, and constant sum check are some of the tests which may be implemented. These flight-critical functions require careful description and analysis.[3]

Test Program Counter--A repeated set of branch-and-return (BAR) instructions are used to test the program counter. A nested set of BARs is used

around the halt instruction so that any error will cause the real-time counter to maintain the halt condition and force the watchdog to time out.

Annunciate Failure Status, Check Sensor Status--The fault flags are analyzed and appropriate warning and emergency panel lights are turned on. These are simple functions.

Calculate Pitch and Lateral Inner Loop Control Laws--The inner loops provide the stability augmentation functions and pilot command augmentation in addition to providing the outputs to follow the outer loop commands. They are the fastest control loops.

Calculate Output Commands--The final shaping and blending of signals to command the actuators is performed.

Select Output Signals--The output commands are exchanged between the computers. Depending on which computers are operating, the signals are compared and the final value for the output is selected. Counters are used to avoid nuisance errors. In some mechanizations models of the actuators are used to determine if the actuators are responding as they are expected. Thus, this flight-critical function may be complicated.

Observations about the Triply-Redundant System

A review of the preceding list of functions suggests several conclusions:

- The functions separate well to allow a carefully structured design for each.

- The interfaces between the functions can be clearly defined. There is no need for circular referencing.
- The functions may be packaged into any executive structure that is appropriate.
- Most of the functions are easy to design and test completely. The logical elements and redundancy management elements may be designed as finite-state automata.[40]

The global consistency of the self-tests and redundant voting is not evident. Each test is added to cover some failure mode without regard to the total system.

There are only a few while-do loops. These loops wait for an external signal to cause synchronization, trap the computation until the watchdog times out to shut down the system, or wait until the real-time counter times up to a particular value. In none of these are there any doubts about termination of the loop. All other loops are indexed for a fixed number of iterations. Thus, the theoretical structure of the software is not complicated. The individual functions may be verified by symbolic evaluation.

The data structure is elementary. There are no problems with overflow of the data base because the numbers of variables, parameters, and constants are fixed. The state of each module is determined by a fixed number of variables. There are no problems with dynamic indexing of arrays.

---

[40] P.J. Denning; J.B. Dennis; and J.E. Qualitz; Machines, Languages, and Computation. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1978.

Design Description

The system is a generalization of an experimental configuration which was documented by its assembly listing and flow charts. The HIPO technique[41] was used to outline the new version. An example of this description is given in Appendix A. We also examined the use of the SRI International formal specification language, SPECIAL.[42] This discussion will be expanded in the subsequent section on tools and techniques. In the next subsection we turn to an outline of a hypothetical distributed configuration.

DISTRIBUTED CONTROL

Figure 4 illustrates four configurations of multiple computers:

- The redundant parallel structure
- The hierarchical structure
- A pipeline structure
- A semi-autonomous structure

The boxes represent computing facilities. Each may be composed of multiple computers in one of the four structures. The triply-redundant system of the previous subsection is an example of the first class. In the hierarchical

[41] H. Katzan (Jr.), Systems Design and Documentation: An Introduction to the HIPO Method. New York: Van Nostrand-Reinhold, 1976.

[42] O. Roubine, and L. Robinson, "SPECIAL Reference Manual," Third Edition, Report No. CSG-45, Stanford Research Institute, Menlo Park, California, January 1977.

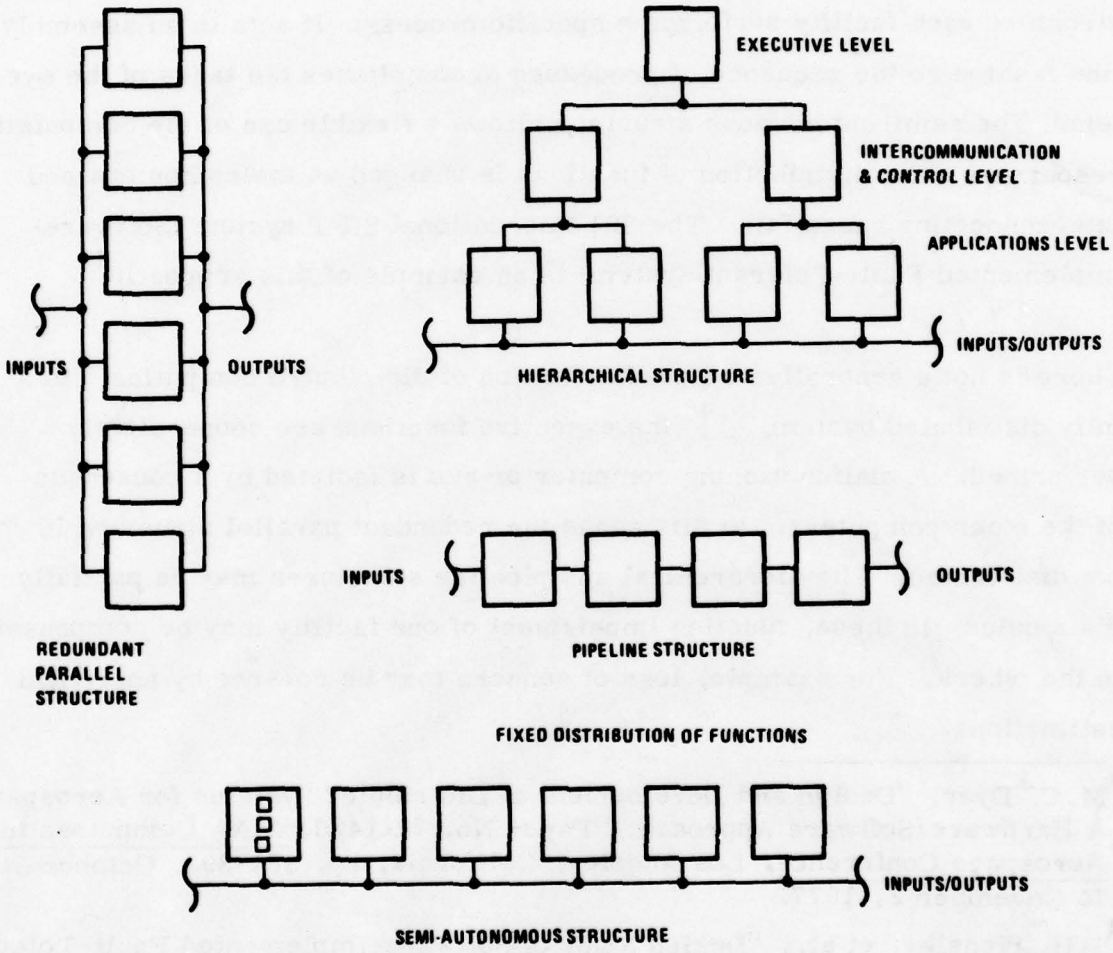


Figure 4. Structures of Multiple Computers

structure executive control originates at the top level, and communications are restricted according to the hierarchy.[43] Many chemical process systems are configured in this manner. The top executive optimizes the productions of the units. The unit control designates the set points of the individual loop controllers for flows, temperatures, pressures, and so on. In the pipeline structure each facility performs a specific process. It acts in an assembly line fashion so the sequence of processes accomplishes the tasks of the system. The semi-autonomous structure allows a flexible use of the computational resources. The distribution of functions is changed as the computers and interconnecting buses fail. The SRI International SIFT system (Software-Implemented Fault-Tolerant System) is an example of this approach.[44]

There is not a generally accepted definition of distributed computing. In a fully distributed system,[45,46] the executive functions are cooperatively performed. A malfunctioning computer or bus is isolated by a consensus of the other computers. In this sense the redundant parallel structure is not distributed. The hierarchical and pipeline structures may be partially distributed; in these, function impairment of one facility may be compensated in the others. For example, loss of sensors may be covered by analytical estimations.

---

[43] M. C. Dyer, "Design and Development of Distributed Systems for Aerospace: A Hardware/Software Approach," Paper No. 77-1450, AIAA Computers in Aerospace Conference, Los Angeles, California, pp. 387-393, October 31 to November 2, 1977.

[44] J. H. Wensley, et al., "Design Study of Software-Implemented Fault-Tolerant (SIFT) Computer," SRI International, Menlo Park, California, June 1978.

[45] P. H. Enslow, "What is a 'Distributed' Data Processing System?," Computer, Vol. 11, No. 1, pp. 13-21, January 1978.

[46] E. D. Jensen, "The Honeywell Experimental Distributed Processor--An Overview," Computer, Vol. 11, No. 1, pp. 26-36, January 1978.

Application to Flight Controls

Distribution of computing tasks for flight controls is attractive because of:

- Enhanced fault tolerance
- Increased survivability
- Opportunities for improving the performance/cost for the life-cycle of the system

Since the functions for flight controls are naturally partitioned, the system is easily configured into a distributed arrangement. However, the system design and analysis become complicated. For example, there are many strategies of allocating and controlling the interconnecting buses,[47,48] and the analysis of failure effects is difficult. For our purposes, the major distinction is whether the functions are fixed in specific computers or, as in the SIFT system, computed by different groups of computers as the failure status of the system changes.

Generic Configurations

The controller for the F-18 is comparison-monitored with quadruplex computers. It reverts to a direct electrical mode upon complete failure of the

---

[47] P.H. Enslow, "Multiprocessor Organization--A Survey," ACM Computing Surveys, Vol. 9, No. 1, pp. 103-129, March 1977.

[48] C. Wise, and J. Bain, "Distributed Processing for MIL-STD-1533A, ASD-TR-78-34," Proceeding of the Second AFSC Multiplex Data Bus Conference, Dayton, Ohio, 1978.

digital system.[49] This configuration is modified in Figure 5 to postulate a distributed system for our study. Six microcomputers, C1 to C6, are used to perform the global executive and control law functions. As microcomputers are lost, the system reconfigures itself through a strategy incorporated in the global executive. With no failures, this function is performed by computers C1 and C2. After four failures, the global executive and the flight-critical functions are calculated in the remaining two computers.

In Figure 5, the redundancy levels are illustrated by the multiple arrowheads. The bus terminals, T_1 to T_{22}, contain microprocessors. Dynamic bus allocation allows the microcomputers to time-share the buses. The pilot inputs are fully independent and quadruply redundant.

The functionality may be partitioned so that the functions are fixed for a group of microcomputers. An example is shown in Figure 6. A group of triply-redundant computers selects signals and monitors the sensors, a second group computes the control and logic functions, and a third group produces the outputs and monitors the servos. Each group must perform its own failure management functions.

Observations on Distributed Systems

Verification of a partially distributed system in which the functions have a fixed assignment will be no more difficult than verification of a generic

---

[49] D.R. Katt, and P.A. Raymont, "The Flight Control Computers of the F-18 Electronics--Flight Control," Second Digital Avionics System Conference, Los Angeles, California, November 2-4, 1977.

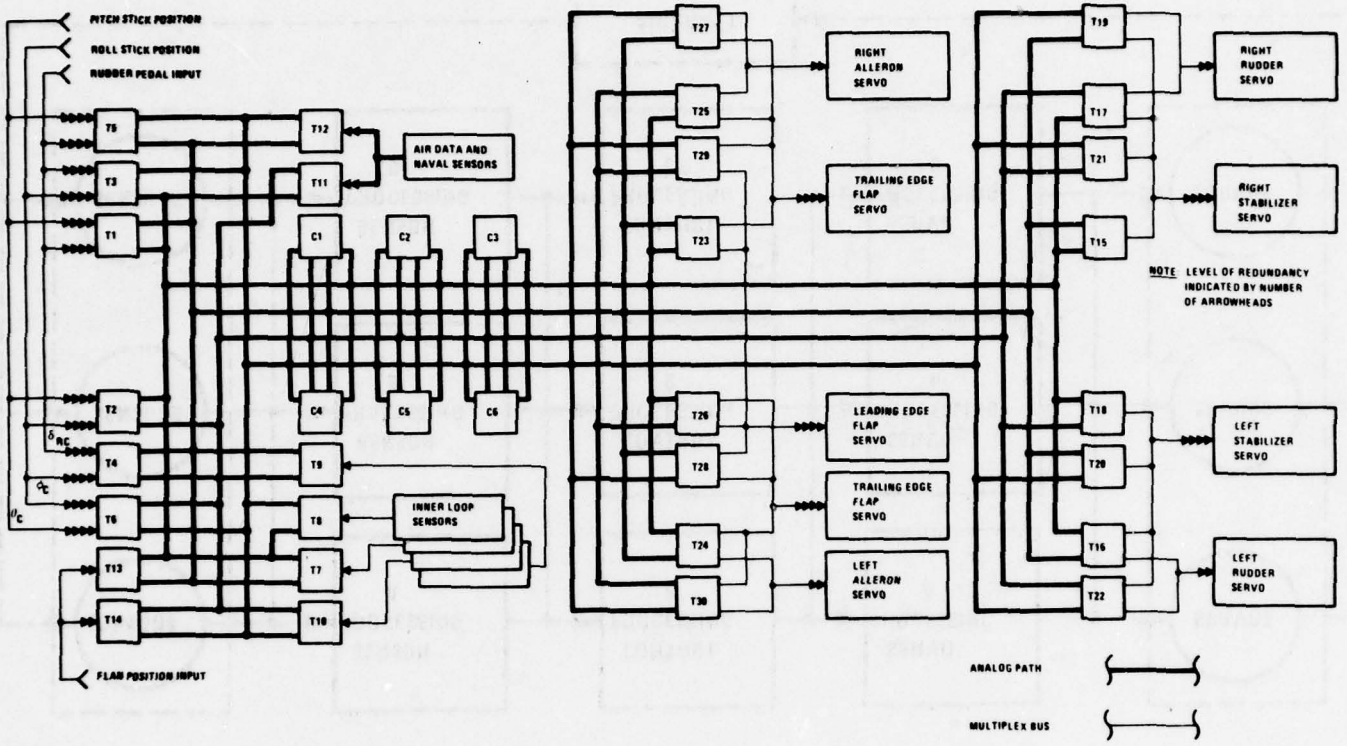


Figure 5. A Fully-Distributed Architecture

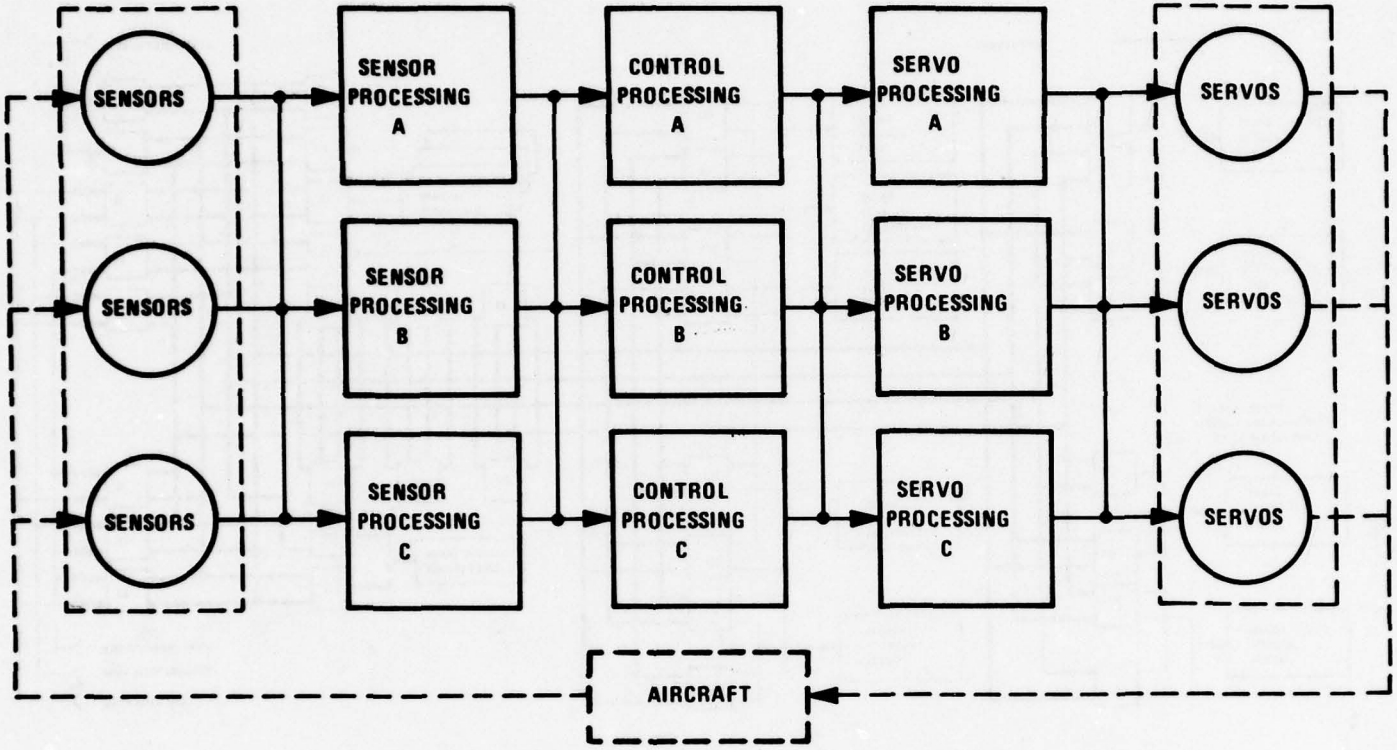


Figure 6. A Partially Distributed Architecture

triply-redundant system. The difficult problems are again in demonstrating that the synchronization and failure management functions are correct and in providing an adequate analysis of failure effects. However, the descriptions and analysis required for fully distributed systems appear to be problems of an additional order of magnitude. SRI is confident that they will provide these proofs for their SIFT system.

FUTURE FLIGHT CONTROL SYSTEMS

The functions for flight control will undoubtedly become more complicated. The power of the computer has made many advanced control techniques practical, such as:

- Multivariable and direct lift control
- Flutter suppression
- Dynamic adaptation to flight conditions
- Battle damage assessment and reconfiguration
- Sensor redundancy by analytical calculations
- Sensor blending and estimation of the dynamic state

These techniques will find use in very high performance military aircraft systems. The computations are orders of magnitude more involved than those for the functions of the triply-redundant system described previously. Their verification will be examined for each case. However, the lengthy analysis which is required to demonstrate the engineering designs will also provide verification of the software implementations.

SECTION III

TOOLS AND TECHNIQUES

The many aids to verification are described in this section. The use of these for flight control software is considered in the next section. The tools and techniques go far beyond methods for testing the code.

A tool is a computer program that performs some task which would otherwise have to be done by hand.[7] Tools may be classified as static or dynamic. Static tools examine some aspect of the specifications, designs, or code without executing the code of the software being inspected. Thus, a static tool is the set/use checker that checks that a variable is given a value before it is used; or if a variable has been defined and given a value, it is subsequently used. A dynamic tool performs some function to aid in testing the program when the program is actually executed. A timing analyzer that monitors and records the execution time of functions and subroutines is a dynamic tool.

A list of tools is provided in Table 9. These are described in Appendix B. The static tools in Table 9 have been grouped into a list of those which examine a specific property and a list of those which examine general or more extensive properties. The set/use checker is listed as a specific tool while a symbolic evaluator, a tool which automatically reconstructs the Boolean or algebraic equations relating the outputs to the inputs, is listed as a general tool. Many of the static tools examine global properties: those related to the program as a whole. For example, the set/use checker may search through much of the program before it can make a determination on a particular variable.

TABLE 9.

TOOLS

Specific Static Tools	General Static Tools	Dynamic Tools
Circular reference checker	Accuracy analyzer	Simulations
Code comparator	Assembly code verifier	• Computer • Hybrid • Test bed (iron bird) • Monte Carlo
Consistency checker	Assertion checker	Test data generator
Cross-reference checker	Documentation and construction systems	Test driver
Data base analyzer	Formal languages with syntax analyzers	Test execution monitor
Flow charter	• Requirements • Specifications • Program design • Program code	Test record generator
Interface checker	Sneak-path analyzer	Timing analyzer
Program flow analyzer	Symbolic evaluator	
Set/use checker	Theorem prover	
Standards checker	Verification condition generator	
Units consistency checker		
Unreachable code detector		

Techniques are the standards and procedures used in the development and maintenance of the software package. In Table 10 we distinguish between those for development and those for analysis or review. The entries are also discussed in Appendix B. The list of development techniques cannot be omitted on the grounds that it is not V&V. Practice has shown that substantial gains in software reliability can be obtained by attention to description, documentation, and systematic development.[50]

A new approach to software development is to use an integrated set of tools which include static and dynamic code analyzers, test and simulation facilities, and documentation aids.[38] Another methodology[51] uses a formal design language based on the constructive approach outlined by Dijkstra, and has an elaborate facility for recording design progress and documentation on a large computer. An integrated approach is also taken for verification systems. An example is the General Research package.[37] In addition to static analysis and testing, verification conditions and symbolic executions are used. These systems show great promise, but currently it is difficult to determine the extent of the error coverage and the completeness of the entire verification procedure.

The general ideas of static, dynamic, symbolic, and analytical verification are discussed in the following paragraphs.

---

[50] G.J. Myers, "A Controlled Experiment in Program Testing and Code Walkthrough/ Inspections," Communications of the ACM, Vol. 21, No. 9, pp. 760-768, September 1978.

[51] D. Boyd; A. Pizzarello; and S.C. Vestal; "Rational Design Methodology," Report No. HR-78-257: 17-38, Honeywell Corporate Technology Center, Minneapolis, Minnesota, June 1978.

TABLE 10.

TECHNIQUES

Development	Analysis
Abstractions and hierarchies to reduce complexity	System concept review
Constructive design approaches	Software design review
Data flow diagram, structure chart	Critical design review
Descriptions	Qualification audit
• Charts -- HIPO -- Flow charts	Checkout testing
	Singularities and extremes testing
• Languages -- Requirements -- Specification -- Program design -- Program code	Integration testing
	Validation testing
	Symbolic execution
• Petri nets	
• LOGOS	
Design guidelines, test guidelines, coding guidelines	
Design standards, coding standards	
Functional capabilities list	
Organization as finite automata	
• Parnas modules • SRI International formal modules	

STATIC ANALYSIS

The tutorial paper by Fairley[25] gives an overview. The information that can be found by static analysis is

- Syntax errors
- Distribution of source statements by type
- Cross-reference maps of identifier usage
- Use of identifiers in each statement
- Subroutines and functions called
- Uninitialized variables
- Variables set but not used
- Unreachable code
- Deviations from coding standards
- Misuse of variables and parameters
- Consistency of references between modules
- Circularities in references between modules
- Consistency of units

These determinations are made by extended syntactic analysis, global analysis similar to the optimization of code by compilers, and simple counting of types.

By analyzing the flow of control from a graph in which the nodes represent the statements of a program and the arcs represent the control paths between

statements, one may detect uninitialized variables, variables which are set but not used, and unreachable segments of code. The graph will also show poor coding style and deviations from standards.

Another graph may be constructed in which the nodes represent the program units and the arcs represent the calls between units. This is very similar to the hierarchy or structure chart. The consistency or circularity of the exchange of variables may be checked. Note that the flow graphs and call graphs are useful in determining test cases for dynamic testing.

It is evident that static tools are specific to a particular language. Analyzers have been coded for FORTRAN, JOVIAL, Pascal, and HAL/S.[52] The only requirement is that the language have a formal syntax so that flow and call graphs may be constructed. The static analyzing capability of the SRI International tools which support SPECIAL is noteworthy. This is non-procedural specification language so there is nothing to correspond to a flow graph but the call graph may be analyzed to prove that all of the interconnections between modules are correct.

There is no question that static analysis is extremely useful and will speed up development. The analysis reduces the need for certain types of tests, particularly the tests to confirm the integrity of the module interfaces. The set/use checks detect the typographical errors in which a new variable is introduced by a spelling error. The checks for the consistency of units,

---

[52] C. Gannon, "A Verification Case Study," Paper no. 77-1443, AIAA Computers in Aerospace Conference, Los Angeles, California, October 31 to pp. 349-369, November 2, 1977.

while elementary, catch many troublesome errors. Any comprehensive system must certainly include static checks in its set of procedures.

ANALYTICAL VERIFICATION

Structured walk-throughs are relied upon heavily to demonstrate the integrity of requirements, specifications, design, and code. They may be classified as informal analytical verification. The effectiveness of walk-throughs depends on the care with which the segment has been organized, structured, and described.

More formal techniques of analytical verification are being developed. Several systems[37] allow assertions to be added which are then formally checked for validity. The while-do constructs create difficulties. They are not significantly used in flight controls so automatic program verification may be done in this limited environment. The cost question governs the issues. The most significant development from our point of view is the work done by SRI International which, when completed, will provide formal verification from specifications down to code.

Specific aspects of assembly-level code may be demonstrated by an automatic verification system.[32] The verifier depends on the assembler and the machine instruction set. Fortunately, the theorem prover section is independent of the language since it only operates on the verification conditions and hence need not be redesigned for each particular machine.

Since the while-do loops in the flight controls are small, isolated, and their termination can be easily demonstrated, symbolic evaluation of the rest will

provide a proof of correctness of the system.[28] Symbolic evaluators are language-dependent and expensive. All that we have read about are experimental. Again there is a cost-effectiveness trade-off. Many of the flight control modules can be very convincingly proved correct by manual symbolic evaluation. For example, mode and switching logic may be demonstrated by calculating the Boolean path conditions for each decision branch. Since the disjunction of all of these predicates at the module exit must be true, the redundancy of the calculations through the flow diagram provides a check that all is well. An example of a proof is given in Appendix C.

The proof that synchronizations work in triply-redundant systems and that interface relations are correct in parallel processing systems is a major concern. Our study is at the beginning level (Appendix A). Certainly, Petri nets may be used to describe the synchronizations of the triply-redundant system.[35] We may be required to look to other methods for verifying distributed software.[36] Another specialized area which should be mentioned is the sneak-path technique developed by Boeing.[53] While the results are impressive for systems validation, the role of sneak analysis in software is not so clearly evident.

Many papers review the utility of formal verification. The review by London[31] gives a balanced account. There can be no dispute of its value for theory and motivation toward clear program design. A senior flight controls analyst commented, "by the time you can do analytical verification you require everything to be obvious." Just so. The problem is to work analytical

---

[53] S.G. Godoy, and G.J. Engels, "Software Sneak Analysis," Paper no. 77-1386, AIAA Computers in Aerospace Conference, Los Angeles, California, pp. 63-87, October 31 to November 2, 1977.

verification into a standard part of software production. Many of the current approaches[51] require so much formal dotting of every i and start with such basic principles that a very heavy overhead is added to the methodology. Proofs must become more credible, less difficult to follow, and less detailed, but this is very hard to do on a machine.

DYNAMIC TESTING

Testing will always be a vital element in the validation process. There is no other way to show that a complicated configuration of hardware and software operates as required. However, as we have seen, the static methods can reduce the tests required and improve the certainty of interpretation for those which are required.

There are several directions for assisting the programmer in his traditional task of testing the code by executing it with particular sets of input data. The basic tasks are:

- Select the test data
- Instrument the code to monitor and analyze the results
- Construct a program to run or drive the code under test and to record and report the results

References 54, 26, and 27 are dedicated, respectively, to discussing the automatic tools for each of these tasks.

---

[54] L. A. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," IEEE Transactions on Software Engineering, Vol. SE-2, No. 3, pp. 215-222, September 1976.

Test Data Generation

The automatic test generators either use symbolic execution and path condition analysis or trace backward through the code to determine how the path conditions evolve. The system described in Reference 54 uses symbolic execution and, in addition to creating test data which execute every program path, it produces a symbolic representation of the output variables in terms of the input variables. It can detect nonexecutable paths and some other special types of errors. The system cannot handle all of the FORTRAN constructs and the generation of test data may be done only for paths that are described by a set of linear path constraints. Neither of these limitations applies to flight control programs. The system translates the source code into an intermediate code before any analysis is performed.

The problem of producing a single algorithm that will yield test data to execute any specified statement in an arbitrary program is unsolvable. There is no such thing. Again arbitrary while-do loops cause the theoretical difficulties. While the automatic generation of test data is still in the research phase for general systems, it appears that a tool for flight controls is now possible.

Test Instrumentation

To instrument a program for test is to insert statements into the code which will monitor and record what is happening inside the program as it is executed. By executing the program over a number of test cases the instrumentation can collect and analyze data on the operation of the program and on the thoroughness of the set of tests. A basic instrumentation is to insert

counters at points in the program to determine the number of times a particular segment of code is executed. Thus, the requirement that each edge of the flow chart be traversed can be demonstrated if the counters have been placed correctly.

The instrumentation can be strengthened by having the probe do more than count. Assertions may be checked or expected values may be compared. To save test time, the probe or monitor could abort the program with an error message if the program enters an unexpected branch. The range of values of a variable could be monitored and analyzed. Some work has been done on inserting the instrumentation automatically.

Instrumentation is the oldest and cheapest tool to help carry out tests effectively. Unlike the other techniques, not much theoretical depth or extensive programming is needed. However, its expression is generally unique with each programmer who uses it.

Software Test Drivers

Test drivers may range from a harness which executes a segment or module of code to automatic systems in which a test procedure is coded in a special test language. The system prepares a report on the results of the tests. This approach provides a notation for specifying tests, a method for executing the tests and checking the results. It eliminates the need for the separate drivers for modules and subsystems. The system may help catch errors resulting as unintended side effects from program modifications by systematically preserving the original test cases for retesting.

Test Planning and Monitoring

A system for preparing test plans using an interactive question and answer format has been constructed. The system is organized through a tree structure numbering system: plans are prepared from terminals with output available in various sorts. Provision is made for moving forward test plans from one release to another (enhancements for one release become regression tests for the subsequent release). The system provides for consistency of planning throughout the test organization and archival storage of planning documentation. Another system provides for on-line monitoring of test status. The results of test sessions are entered from terminals indicating test cases executed, results, error report identification, edit level used, and date. Reports are produced summarizing test case results by test plan number, identifying status vs. schedule, lists of error reports, and correlation of specific test cases to functionality checklists are also maintained on the data base. The data base also includes error reports, software development responses, and fix transmittals; it permits correlation among the three as well as identification of the test case causing the error.

Thus, information processing techniques may be used to implement the bookkeeping of testing.

SUMMARY

There are many tools and techniques for improving software reliability and for relieving the programmer of the tedious aspects of the job. Static analysis provides many useful checks beyond syntax compliance. Analytic

verification provides the theoretical foundations and can already provide useful tools for a restricted class of software. The tools to aid tests are important for software shops which handle sequences of similar jobs.

Most of the tools are specific to the language in which the code is written. Since writing the program called a tool is expensive it must have a wide application. It is clear that any shop that will build a set of tools must write the bulk of its programs in a standard language. This is another reason to hope for the success of the DoD language program.[55]

In the next section we consider how these tools and techniques aid in developing software for flight controls by considering methodologies with increasing levels of automation. This section is concluded with some general observations on using the tools.

In general, each tool analyzes a particular property of the design, code, or test. For debugging at the designing, coding or testing stages, the tools are very useful. However, since there are generally implicit assumptions made about the correctness of the other aspects while a specific tool is being used, care must be exercised when interpreting the results for verification. For many of the flight control functions, symbolic evaluation by hand or by machine provides convincing verification. An automatic flow charter can be used to demonstrate that program control structure is according to design. The flight control program may be structured to allow precise verification that the data transfers between functions are correct, and there are no side effects by errors in the scopes of variables. The next section shall contend that the software may be verified with confidence.

---

[55] D. A. Fisher, "The Common Programming Language Effort of the Department of Defense," 1977 Computers in Aerospace Conference, Los Angeles, California, pp. 297-307, November 1977.

SECTION IV

VERIFICATION METHODOLOGIES FOR FLIGHT CONTROLS

The background for the examination of verification procedures for flight controls was reviewed in the first three sections. In the first section the development cycle and its errors were sketched. The functions required of typical flight control systems were outlined in the second section. The new tools and techniques available to help software development and verification were described in the previous section. The use of these tools and techniques and the fundamental problems of verification and validation for flight control systems are considered in this section. To illustrate the design and verification process, four levels of methodologies are outlined as follows:

- Non-machine methodology
- V&V with a set of tools
- Integrated development system
- Formal methodology

The levels are not suggested as finished procedures but are frameworks on which the tools and problems of design and verification may be discussed. The fundamental tasks for V&V are outlined first, then the four methodologies are introduced. These are evaluated by applying several criteria, particularly the software errors outlined in Section 1. The final topic is a subjective review of the reliability and confidence in the integrity of the software which the verification process achieves.

FUNDAMENTAL TASKS

Verification is the process of showing that the software fulfills its specifications and that it is free of all technical errors of software design and construction. Because of the complications of rigorously representing the semantics of the computer constructions and the questions of correctness of the language translator, it is often felt it is impossible to absolutely prove the correctness of general software. Therefore, the verification process can only increase the confidence in the accuracy of the software. However, the simplicity of the computers and software for flight controls leads to a much stronger claim. The verification procedures must assure all errors have been detected and eliminated. This places a heavy burden on the specifications. The validation process must demonstrate that these specifications are complete and consistent, the entire hardware/software system meets its requirements, and it reacts correctly in all situations. The narrow use of the terms verification for proof of the technical correctness of the software and validation for the demonstration that the system performs as intended will be followed in this discussion.

A verification methodology for flight controls may be configured to be rigorous and formal to any level. It may therefore be automated with tools to any desired level. Automation is the final demonstration of the formality and rigor of the approach. While the software may be convincingly verified by many techniques, the validation of the system is much more difficult. Because there are so many possibilities, some form of an inductive argument is explicitly or implicitly used to substantiate the claim that all anticipated situations are correctly handled. It is not a simple task to prove that all the flight control functions, particularly those for failure

management, are consistent and complete. Only experience and very careful analysis can reduce the probability of an unanticipated event occurring with the system. Fortunately, there is a large reservoir of experience in the engineering of flight control systems.

Some of the fundamental tasks for verification may be listed without undue complication. Each function must be shown to be correct, and the data flow between functions must not introduce errors. For each function its inputs, state variables, and outputs must be identified. The skeleton of the program will be said to be technically correct if it is shown that:

1. All state variables are initialized correctly and are correctly preserved for the next cycle of computations.
2. The data transfers between functions are shown to be correct so that outputs are available as inputs where required and the intervention of a third function does not unintentionally alter a transfer.
3. There are no circular references or recursive references.

Condition 2 may be stated more generally that the program should be modular and not susceptible to side effects. Condition 3 is not absolutely necessary for formal proofs, but it simplifies the task considerably and makes the demonstration more convincing. It is certainly needed to allow changes to be made safely.

Numerous items must be checked for each function. For example, all looping or repeating calculations must be shown to terminate correctly, and the indexing of arrays must be shown to be within bounds and correct.

It is common to be off by one or not have the variables for the loops initialized correctly. All singular values or divisions by variables must be handled correctly, and the boundary values on the ranges of inputs must be checked. For general software it has been observed that many problems in the field come from almost correct inputs. The programs process good inputs correctly and handle very bad inputs correctly, but unpredictable results may occur with slightly incorrect inputs. Singularities and incorrect inputs are seldom problems in flight control systems, but overflow and scaling must be watched carefully.

The proofs that the functions perform according to their specifications may vary. This is considered when discussing the four methodologies; the method of proof is chosen to suit the function. For most of the functions for flight controls, input and output assertions may be written. The evolution of the program may be followed by symbolically evaluating the conditions along the paths.

The specifications of the functions may be incomplete or allow conflicting responses between functions. For example, the test for data exchanges between triply-redundant computers may correctly detect a unilateral error in a data bus and shut down the transmitting computer. If inconsistent data have been allowed into the other two computers, the comparison monitors on the outputs may shut down the system, completing the single-point failure. The methodology must be able to uncover these problems.

Numerous details must be checked. A methodology must be able to cover all facets of the work. Management structures are necessary to control the configuration and documentation during design and coding to provide for

subsequent changes. Procedures must be included to assure that no mechanical errors creep into the program as it goes through the many steps to the final hardware system. Much of this may be automated. In this discussion, these managerial details will be omitted.

Validation of the system cannot be done as systematically and completely as the verification of the technical correctness of the software. The difficult problems, completeness of requirements, correct mathematical models of the physical processes, and the vast number of situations with which the control system must cope enter at this point. The validation process must cover it all.

The techniques for validation are:

- Testing
- Emulation
- Simulation
- Failure analysis
- Reliability estimations
- Mathematical proofs

The techniques all have limitations. A thorough validation program uses all of them. Testing, simulation, and emulation are the basic approaches. The fundamental problem is in determining the coverage of all possibilities. Failure analyses come in three forms: 1) failure modes and effects analysis; 2) fault-free analysis; and 3) success path analysis. The failure analyses are supplementary and generally require very lengthy and detailed work. The failure analysis for the 301A computer in the JA-37 system required seven

man-years. Markov models and other probabalistic representations are useful for the study of configurations. The inadequate statistical descriptions of the faults limit the quantitative results which are obtained by these means. Mathematical proofs will suggest the framework needed for establishing the validity of systems, but they will not supplant the need for the other techniques. Improving the validation methods is a current area for research.

The fundamental tasks may be summarized as:

- Verification
  --Show each function is correct
  --Show the program skeleton is correct
- Validation
  --Demonstrate consistency for normal events
  --Demonstrate consistency for a class of anticipated failures
  --Inductively generalize to attempt to cover unanticipated failures

In the following discussion on the four levels of methodology, the focus is on how the tools help the verification process.

FOUR LEVELS OF METHODOLOGY

To organize the study of how the new tools and techniques fit into the development of flight control software, three dimensions are considered. The first is the stage of software development, the second is the type of flight control function being verified, and the third is the level of automation included in the methodology. Four steps are studied in this dimension. The first reviews the verification of the generic triply-redundant system of Section 2 by hand. From this it can be projected how the tools can help reduce the work and add confidence to the verification. This yields the second step,

V&V with a set of tools. The third step is a software development system. It provides an integrated set of tools to aid the programmer in the entire job of designing, coding, and verifying. The final level is the completely formal approach in which each facet of the requirements, the language, and the machine is incorporated into a hierarchical structure of abstract machines. Verification of the software proceeds automatically.

The method for verifying a flight control function may be chosen to be particularly suited to the function. The tasks required of redundant flight control computers are classified as:

- Executive functions
- Synchronizations
- Logical functions
- Control law calculations
- Test functions

The executive functions perform initializations, branch to rate paths, and report the failure status. The synchronizations keep the redundant computers in step by frames and allow the exchange of data. The logical functions monitor sensors and outputs and make the proper selections from redundant data, manage failures, and calculate the flight control mode and switching logic. The outerloop and innerloop flight control laws produce the outputs to drive the actuators. The test functions are a collection of self-tests on the hardware for each channel. The data exchanges, power supply, output hardware, and many elements of the computers are tested. In addition to the method of verification of the particular functions, the manner of their

description, the way in which tests are to be constructed, and the way in which the consistency of the system is demonstrated must be chosen.

The success of this approach depends upon the care and diligence which is devoted to the reviews. A measure of this is the clarity, consistency, and accuracy of the documentation. In the review it is assumed that the work is checked by a knowledgeable, skeptical reviewer. Poor communication makes this very difficult. This approach cannot establish correctness, even at the verification level, because it has no inductive step.

The four methodologies address Stages 3, 4, and 5 in Table 11. The new tools and techniques provide the most help at this point. The validation in Stages 5 through 8 are assumed to follow traditional lines.

A Non-Machine Methodology

A non-machine methodology could alternatively be called a non-automated methodology. An outline is provided in Table 12. The headings of the table are the stages of development vs. the task of function to be considered. The table entries are the tools or techniques which are suggested. Descriptions of these are given in Appendix C. The design is described by HIPO charts. The inputs, states, and outputs of each function at the most detailed levels of the hierarchy chart are easily defined. After checking these, the inputs, states, and outputs of the level which calls the lowest level are verified, and the data exchange is checked. In this manner the data flows in the entire program may be reviewed to establish the correctness of the function calls. An analysis of data flow will help establish a basic level of consistency between the functions. From the HIPO charts, decision tables may be used to set up tests for checking the code. From the data flow analysis, integration

TABLE 11.

REVIEWS AND TESTS

Stage	Reviews	Tests
1. Requirements	Systems requirements review	--
2. System design	Software concept	--
3. Software design	Preliminary design review	--
4. Coding	--	Module tests
5. Integration	Critical design review	Integration tests
6. Qualification	Qualification Audit or functional Configuration audit	Validation tests on operational hardware
7. Installation	Physical configuration audit and Formal qualification review	Validation tests On iron-bird simulation
8. Maintenance	Change reviews	Re-validation tests

tests may be constructed. These tests will provide confidence that the program control structure from the design stage has been correctly rendered by the code and no typographical errors have occurred.

The design of the flight control functions may be verified following Table 12. A review will suffice for the executive program. Confidence in the synchronization scheme may be established by constructing tables of events and Petri nets as illustrated in Appendix A. The logic functions and self-tests may be checked by symbolic evaluation. For these, input and output predicates

TABLE 12.

A NON-MACHINE METHODOLOGY

Functions	Design	Coding	Integration
General description	HIPO	Assembly language	Assembly language
Test construction	Decision tables	Tables of: Inputs/outputs Constants States Module references	Integration tests
System consistency	Data flow analysis and finite-state analysis		
Control functions			
Executive	Review		Test rate-path timing
Synchronization	Table of events	Review	Integration tests
Logical	Symbolic evaluation	Decision tables	Integration tests
Control laws	Review	Frequency responses	Integration tests
Self-tests	Symbolic evaluation	Decision tables	Integration tests

may be written, and the progress of the calculations may be followed by writing the path predicates. An example is given in Appendix C. Only a review of the control laws, with special emphasis on the manner in which the laws are switched, is needed at the design stage. The code is written in assembly language. The executive may be exhaustively tested, and the timing relations may be checked after integration. A review of the code to check the agreement with the design will be adequate for the synchronization. For the logic and self-tests, decision tables written from the code may be checked against the previous symbolic evaluation or a symbolic evaluation may be developed directly from the code for comparison.

Verification of the control functions after integration will continue to rely on testing. To be sure, with the increased attention to design and coding, the verification is directed more to typographical and clerical errors. Because of the non-machine approach, testing will be important for rendering confidence. The accuracy of the control law calculations may be checked by frequency response measurements at any stage. For most flight control applications these computations represent analytic functions of simple additions, subtractions, and multiplications. An occasional division by a variable is found in a schedules gain parameter. The frequency response check will show that all signs and operations are correct. Scaling, overflow, and nonlinearities such as break-outs and limiters are checked during validation simulations.

A complete program of analysis and tests will verify that the software is technically correct and free of errors. The statement that testing can only show the presence of errors only weakly applies to the simple and uncomplicated code that is used in flight control systems. This, of course, assumes that the code has been carefully inspected so there are no ingenious, obscure,

or non-standard constructions in it. The amount of calculations done during simulation of a closed loop control system is very large. A careful test program will establish a very high level of confidence in the validation of the normal operation of the system.

V&V with a Set of Tools

It is possible to formalize the verification procedures from the previous discussion. This means that precise definitions and specifications are given for the concepts and procedures. The next step in the development of the non-machine methodology must accomplish this. Once formal constructs have been chosen, they may be implemented using software tools. Another approach is to review the tools and techniques now available and determine how they may be utilized for verification.

A set of tools may be chosen to do much of the work and to increase the level of confidence in the results. Tools are expensive, hence they must have wide application. Also most tools apply to software written in a particular language. In the approach outlined in Table 13, it is assumed that the program design language has a syntax that may be analyzed and the HIPO description has been formalized so that static verification of function references, input/output consistencies, and circularity of references may be checked by machine. The consistency of the functions is checked by a machine version of the graphical data flow analysis. The coding is done in a high-order language which has syntax checks and the capability for extended static analysis. At the function level this would include verification of the program control structure by the flow chart generator, programming standards analysis, and a check of the consistency of the units. After integration, the static tools would again verify the program control structure by generating the flow

TABLE 13.

V&V WITH A SET OF TOOLS

Functions	Design	Coding	Integration
General description	Program design language and HIPO	High-order language	High-order language
System consistency	Data flow analysis	Flow chart generation	Set/use checks
Static analysis	Syntax checks Input/output verification	Standards check Units consistency	Unreachable code Interface checks Circular references
Tests	Decision Tables	Instrumented tests Test data set Record generator	Instrumented tests Test data set Record generator
Control functions			
Executive	Symbolic evaluation	Exhaustive tests	Timing analyzer
Synchronization	Table of events	Exhaustive tests	Monte-carlo simulation
Logical	Symbolic evaluation	Symbolic evaluator	Monte-carlo simulation
Control laws	Review	Simulation	Simulation
Self-tests	Symbolic evaluation	Symbolic evaluator	Monte-carlo simulation

charts, checking for unreachable code, and verifying initialization and introduction of variables by the set/use checker. The correctness of the technical skeleton of the program is assured by these machine aids.

Tests may again be selected from analysis with decision tables. The test facility may use tools to instrument the tests, call on stored test data, and generate test records. All of this is especially useful for changes and modifications during the maintenance stage. Monte-carlo tests will add confidence that the tests cover more than hand-selected cases for which the design is known to be correct.

At the design stage the executive, logical, and self-test functions would be verified by hand analysis since it is unlikely that a symbolic evaluator would have been constructed for the program design language. However, such a tool would be available for the code in a standard high-order language so that the code for the logical and test functions would be verified automatically. It is probably easier to verify the executive and synchronization functions by exhaustive testing and to wait for verification of the control law calculations until a simulation is available than to use formal tools. The simulation analysis is needed to check the control engineering aspects, regardless of the software verification procedures.

An Integrated Development System

Hypothesize that a general software development system has been constructed and that it is similar to the NASA MUST package.[38] The system is intended for general avionic software development so the tools are not chosen specifically for flight controls. While the advantage of choosing tools is lost, the

system provides the programming support for editing, managing data, maintaining an account of the configuration and of the documentation, and generating reports. There are static and dynamic code analyzers, a flow charting system, and facilities for test case generation, test coverage assessment, and simulation. However, a symbolic analyzer and tools for analyzing module interfaces and connections are not included.

The methodology is outlined in Table 14. The simulation facility allows the program to quickly produce a digital simulation. Step, ramps, functions, and noise are available as inputs. This extensive capability for simulation may be used for a large share of the verification needs. It is useful to keep the results of the simulations for comparison with the actual flight code during the validation stage. For design, the methodology reverts to hand analysis. Since a high-order language is used, the syntax and extended static aids are available. Flow chart generation is used to show that the control structures of the individual functions are correct, but the analysis of the functional interfaces must be done by hand. This forces more reliance on the integration test program.

A Formal Methodology

The ultimate approach to development is a completely automated integrated methodology. The hierarchical approach of SRI-International is a model. The stages of design, code, and integration are changed to specification, program, and implementation because the software is already integrated at the specification level. The functions are still grouped in modules, but the internal and external consistencies are verified at the specification stage. The verification methodology is outlined in Table 15.

AD-A076 021 HONEYWELL SYSTEMS AND RESEARCH CENTER MINNEAPOLIS MN F/G 1/3
DIGITAL FLIGHT CONTROL SOFTWARE VALIDATION STUDY.(U)
JUN 79 E R RANG , M J GUTMANN , D B MULCARE F33615-78-C-3605
UNCLASSIFIED 79SRC18 AFFDL-TR-79-3076 NL

2 OF 3
AD
A076021

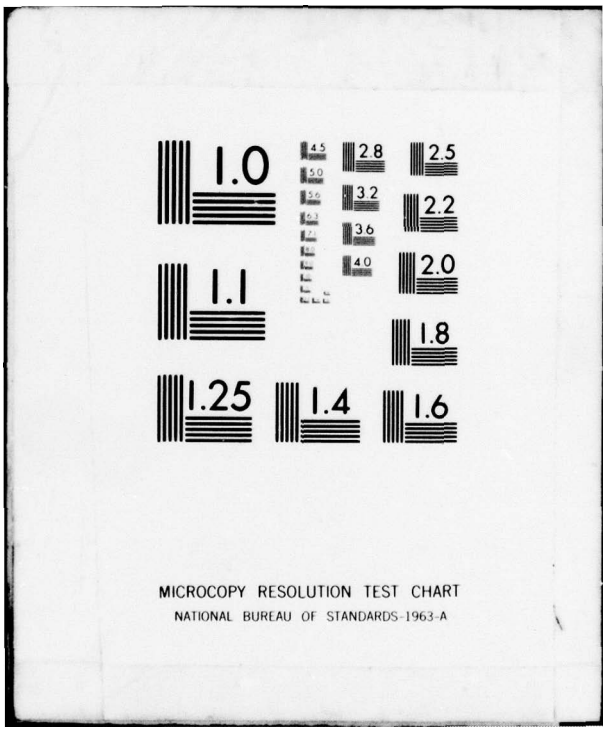

1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

TABLE 14.

AN INTEGRATED DEVELOPMENT SYSTEM

Functions	Design	Coding	Integration
General description	HIPO	High-order language	High-order language
System consistency	Data flow analysis and finite-state analysis	Tabular input/output analysis	Interface analysis
Static analysis		Flow chart generation Units consistency Set/use checks	Set/use checks Unreachable code
Tests	Decision tables	Instrumented tests	Instrumented tests
Control functions			
Executive	Review	Exhaustive tests	Integration tests
Synchronization	Table of events	Exhaustive tests	Integration tests
Logical	Symbolic evaluation	Decision tables	Integration tests
Control laws	Review	Frequency responses	Integration tests
Self-tests	Symbolic evaluation	Decision tables	Integration tests

TABLE 15.

A FORMAL METHODOLOGY

Functions	Specification	Program	Implementation
General description	Formal specification language	Intermediate procedural language	High-order or assembly
System consistency	Finite-state analysis		
Static analysis	Set/use checks Interface verification Cross-reference analysis Circular reference checks		
Formal verification	Assertions checked	Program functions, verified with specifications	Implementation, verified with program
Tests			Minimal system-level tests
Control functions			
Executive Synchronization Logical Control laws Self-tests	Specifications verified to requirements by hand analysis and review		Frequency response tests

The specifications are written in a formal non-procedural language. All of the syntax checks and static tools are available to prove consistency within the system. The specifications must be verified to correspond to the requirements. The relation between the requirement and the specifications and its assertions can only be verified by review; formal requirements have not been assumed. Therefore, system level tests are still necessary for validation in the last stages of development. The consistency of the failure management design must also be demonstrated by other means.

The formally specified functions are called by a program constructed in an intermediate language. This forms the top-level interface and is considered an implementation on an abstract machine. The next step is mapping to a less abstract machine. After a series of reductions of abstractions, the actual machine is reached. All of these levels of abstract hierarchy can be proven by automatic methods. For flight controls, only two steps have been found useful.

In this methodology, there is no need to test code. Validation testing and the simulations to check the engineering of the control laws and other system details are necessary.

Comparison of the Methodologies

These four methodologies illustrate the directions in which design and verification may proceed and how the tools and techniques may be used for flight control systems. The first three can be amplified to be more formal with procedures and definitions carefully formulated. The fourth is already formal. The procedures may be automated as the level of formality is increased. For example, the proof that the skeleton of the program is correct

may be automated with only a small increase in formality of the HIPO charts.[11] The non-machine methodology may be formalized to provide a mathematical analysis which is as convincing as the care of organization and exposition that is put into it.

Of the large number of aids for the programmer, the static tools are the most useful for debugging designs and code. The dynamic tools of instrumentation and test management greatly improve the execution of the testing program. Most tools verify a particular attribute, such as the set/use of variables and consistency of units, and thus help the programmer in debugging. They add confidence to the work but not absolute certainty. Some of the tools, such as the symbolic analyzer, will provide complete verification of most aspects of the flight control functions. Even here, not everything is covered. Overflow, numerical accuracy, and interfaces between functions may not be checked. Thus for a high level of verification by machine, a set of tools must be used. The advance methodologies, which will be totally formal, may be able to provide a complete machine verification of all aspects of the development; but it will be some time before these systems are generally available.

Automation introduces discipline into the development process and reduces the necessary tests to check for manual errors and omissions. The sequence through the four methodologies shows the trend to more routine and less arbitrary choices. Less is left to the thoroughness of the programmer and the management of the development cycle. The integrated development system and the formal methodology provide configuration and data management. In the formal approach, changes cannot be made without returning to the proper entry point in the development cycle. The formal approach, in requiring

non-procedural specifications for the modules, also forces a systematic structure on the design of the program. The structure has a significant bearing on verification.

Design for Verification

Experience has shown that software quality is greatly improved when the need for verification is kept in mind during the design process. To facilitate explanation and review, care must be devoted to the structure and documentation of the design. The concept of a finite-state machine will help structure the design of the software implementation of a function when it is applicable. Fortunately, most of the logic and test functions may be interpreted as finite-state machines. An example of this approach is provided in Appendix D.

The state is represented by a fixed set of variables that determine the internal condition of the machine. For each computing cycle, a set of input values is read. The output is computed from these inputs and the current values of the state variables. A transition to a new state is then made depending on the inputs and the current state. The design of the implementation of a function may be accomplished along the following lines:

- Define the states of the machine
- Define the events which change the states
- Construct a table showing how the events cause the transitions of the states
- Construct the functions that interpret the inputs as events
- Construct the functions that yield the outputs in terms of the inputs and the states

One may proceed by successively refining the machine by adding more details and design decisions to the implementation.

The example in Appendix D illustrates this process for a signal-selection algorithm within the redundancy and failure management subsystem of the generic flight control system. It should be possible to follow the same approach for the other functions in the subsystem. If the approach can be extended to the total failure management subsystem, then that complex package can be easily verified.

Most of the functions for the flight control system may be implemented in this manner. The design fits naturally into the HIPO description. The concept of an event that causes a transition allows a concise definition of how the function is modeling the physical system.

CRITERIA FOR V&V

The previous discussion of how tools and techniques may be used for flight controls was general. In this subsection the transitional goals for V&V are reviewed.

The principal qualitative objectives for verification are to:

- Identify and eliminate errors at each stage of the design and implementation process
- Lower development costs by reducing the extent of testing and simulation
- Improve the structure and robust character of the software design and improve the surety of making changes; produce a more modular system

- Aid communications and understanding between programmers, project office, and users
- Help communications and understanding between programmers, project office, and users
- Help programmers in tedious, routine tasks

Other considerations are the ease in learning how to use the item or the difficulties in operational use of the item in the total development methodology. This touches on the overall cost/effectiveness question which is central in constructing the V&V methodology. The methodology must enhance and enforce the designs to be easy to test, modify, and document. Designs must be easy to change, expand, or contract.

Measures which are more quantifiable are:

- The cost of constructing and using the tool
- The restrictions of the designed software to specific languages or representations
- The stages of the development process in which the tool or technique is effective
- The types of flight control functions that are easily handled
- The categories of software errors which are detected or eliminated by the methodology

The use of criterion of coverage of errors categories follows the work reported by Thayer, et al.[7] It is an excellent and comprehensive study.

Section 4.7 will be reviewed in some detail. The considered techniques were divided into preventive and detective classes as follows:

- Preventive
  - --Design standards
  - --Coding standards
  - --Design inspection
  - --Code inspection
- Detective
  - --Functional capability list
  - --Data singularity and extremes testing
  - --Simulation testing of algorithms
  - --Integration test
  - --Requirements or validation test

The tools examined were:

- Preventive
  - --Simulation
  - --Design languages
- Detective
  - --Code standards auditor
  - --Units consistent analyzer
  - --Set/use checker
  - --Compatibility checker

Other tools discussed are:

- Dynamic path analyzer
- Test data generator

- Test case execution monitor
- Generalized data base construction tool
- Data base comparator

Errors found during the development of a large software project were analyzed and assigned to error categories. A table of the percentage of errors in each category which were judged to be susceptible to the technique was presented. An important observation is that the judgment was seldom made that the technique covered 100% of the errors of a particular error category. Therefore, the criterion is less quantifiable for evaluating V&V techniques or care must be used in choosing the appropriate categories.

The table gives the following summary of the percentage of errors which would be caught by the techniques:

Design standards	28%
Coding standards	26%
Design inspection	58%
Code inspection	63%
Functional capability list data singularity and extremes testing (actually total useable data)	61%
Algorithm simulation	8%
Integration test	46%
Requirements test	46%

The summary for the tools is:

Simulation	20%
Design languages	32%

Code standards auditor	20%
Units consistency analyzer	2%
Set/use checker	14%
Compatibility checker	10%

A strong conclusion was made. Additional testing should have been done at the function level prior to integration testing. However, the combination of validation, acceptance, system integration, and operational testing was effective in catching the errors that should have been detected earlier.

Analysis of data from a subsequent project verified the qualitative conclusions on utilizing the tools and techniques. However, missing logic errors and data initialization errors continued to be predominant in the second project.

There is no similar data for a digital flight control development. On the JA-37 project, the design guidelines were set very conservatively but the programmer/analysts fixed their own bugs. No records were kept. While there were some design changes to augment functions, modify control laws, and such, after the final acceptance checks were made, no errors in the code were ever detected.

COVERAGE OF SOFTWARE ERRORS

The errors for each stage of development were listed in the Introduction. These are summarized in Table 16. The errors of Stages 1 and 2, the system stages, are only partially covered by the software verification methodology. Better software techniques will induce better system techniques.

TABLE 16.

ERRORS

	Stage 1 System Requirements	Stage 2 System Analysis and Design	Stage 3 Software Analysis and Design	Stage 4 Coding and Checkout Testing
Errors in Inputs	System requirements are: • badly stated • changed from those written • incomplete or inconsistent • overly restrictive	System requirements are: • misinterpreted • poorly documented Data defining the system is incomplete Descriptions of the hardware and the peripheral equipment are inaccurate or incomplete	Lack of coordination between Stage 2 and Stage 3 Poor documentation of the hardware Inadequate data on the configuration	Incomplete or erroneous design specifications Poor documentation of the hardware
Errors in Tasks	System requirements are: • misinterpreted • poor[illegible]ocumented C[illegible]tion of the system is no correct Size of the computer is inadequate Hardware interfaces are unclear Validation plan is not adequate	System configuration is wrong: • poor hardware/software trades • awkward executive control structures • faulty computer synchronizations Documentation of the specifications is incomplete Poor selection of computer language Software test procedures are poorly designed Concept review is done badly	Functions are partitioned incorrectly Software organization is insecure Poor utilization of: • software standards • guidelines • programming methodology Incorrect algorithms Documentation is incomplete and inadequate Preliminary design review badly done	Coding errors: • misinterpretation of language constructs and hardware operation • overflow, scaling and approximation • self-modification of instructions • sequencing and branching errors • incorrect loop structures • loss of index or state data • mishandling of singular and critical values • errors in computing equations • incorrect values for constants • unreachable code • uninitialized variables Code not reviewed Typographical errors Incomplete testing of modules Incorrect implementation of design specifications Poor documentation

Stage 1 Errors

The major errors of this stage are concerned with the communication of the requirements of the project. The errors in the tasks come from this misunderstanding. Fortunately, for flight controls, both the Air Force and vendors have experience in this transaction and generally know where the problems may fall. On the other hand, many preliminary decisions are not subsequently reviewed in the press of the later phases.

Scanning the tables of tools and techniques, it is noted that few things help out in this stage. The formal languages for requirements have not been carried very far and are a research topic, The major item is beginning the functional capabilities list for the system. This will form the basis of the validation testing program and be carried along as documentation through the entire development. Guidelines for requirements may be established to help determine if an item is genuinely a requirement. Carefully documenting the requirements preparing the functional capabilities list will help show where the requirements are poorly stated, incomplete, or overly restrictive. Unfortunately in many cases, the requirements are warmed over mechanically from previous projects, and the attention necessary to point up problems is not invested here.

Stage 2 Errors

The system concept review is traditionally the major V&V technique relied upon in this stage. However, the functional capabilities list for the software represents a high level in the approach to limit complexity by abstractions and hierarchies. Data flow diagrams may be sketched. A preliminary

partitioning of functions into modules may be done. Petri nets and simulation may be used to study and verify the synchronization schemes. General documentation and general information systems for software construction may be called into use. The success of this stage depends largely on the diligence with which the requirements specification document is prepared.

Stage 3 Errors

The analysis and design of the software begins. The tools and techniques now have more to offer. The traditional approach to V&V relies heavily on the software design review, but many things help make this presentation more exacting. The next division of hierarchy partitions functions into the computer program components. Design guidelines and design standards help determine the general practice. Descriptions by HIPO charts and program design language allow the documentation of the design to be as complete and precise as the designer is willing to make it. Organization of the flight control functions as finite automata helps in defining the state and its transitions, subfunctionalities and the manipulation of data inputs to the automata, and the packaging of the results as outputs.

The formal specification languages may now be used. The syntax checks, consistencies within each module, references between modules, and the circularity of references between modules may then be verified by automatic tools. The specific errors of the Tasks of Stage 3 follow.

Functions are Partitioned Incorrectly--The worst offense is circular referencing between functions. This makes verification difficult if not impossible. It also renders the program difficult to change without introducing strange errors. Other poor partitioning produces awkward and obscure structures.

The JA-37 software has several circular references of a fundamental nature, not just packaging errors.

Structured design and the HIPO description reduce the occurrence of these errors. The data flow between modules can be analyzed by hand assembly of the input/output quantities into tables or it can be checked by formal tools.

Software Organization is Insecure--Errors of this type result in unwanted side effects or inappropriate responses. In the generic system, the utility program may be entered from the executive function. Interlocks must be provided to prevent accessing this function in flight.

Again, good structured design will lessen the risk of these errors or point up the need for system-wide safety features. System design errors are difficult to catch by formal methods.

Poor Utilization of Software Standards, Guidelines, and Programming Methodology--This is primarily a management problem although some deviations from good practice can be detected by static analysis tools. It is our judgment that formal features need not be built into the methodologies for flight controls. Advances in the pedagogy of computer science will also decrease these problems as the younger people become active in this work.

Incorrect Algorithms--The requirement of careful descriptions and walk-throughs with knowledgeable colleagues will reduce errors in writing designs for the algorithms. Organizing logical functions as finite-state automata will also help. This requires the designer to define all the possible states and events which cause transitions. The result is a precise and certain top-level design.

Documentation is Incomplete and Inadequate--This management problem is aided by the HIPO approach and the awakening awareness of programmers to the vital importance of documentation if the work is to be understood and if changes are to be made easily. This is the usual place to short-cut when time and funds get tight. It is incredible that much software still goes out as poorly commented assembly code and flow charts.

Preliminary Design Review Badly Done--The review is the traditional method for verifying the design. It is doubtful that a formal meeting can accomplish the task. This approach is a vestige of hardware development in which the correctness of design can be adequately checked by general discussions. For software, the details are essential. In any case, the documentation presented in the review must be thorough. The verification methodology must require that the walk-through review be done completely for it to establish any level of confidence.

Stage 4 Errors

The majority of the tools apply after the design has been coded. There are one or more tools which will cover the coding errors in the Stage 4 list. Many of the flight control functions fit into modules which may be tested exhaustively. Few functions have critical values or extremes of data which may cause uncertainties. Leaving aside testing, symbolic evaluation, and the methodologies, it is observed that the static analysis functions (set/use checks, unreachable code detection, and units consistency analysis) and the consistency determinations between modules detect many typographical errors.

The specific errors listed in Table 14 for Stage 4 follow.

Incomplete or Erroneous Design Specification and Poor Documentation of the Hardware--The inputs to the coding phase will carry the system design and software design errors. Faulty communication of the hardware functions and interfaces will also hinder the coder. Otherwise the coding will proceed quickly and accurately.

Coding Errors--These have been studied extensively and are the errors most easily detected by the static tools. Reviewing the list, a table of the most appropriate tool may be constructed as follows:

Error	Tool
Sequencing and branch	Flow-charter
Incorrect loop structures	Flow-charter
Unreachable code	Flow-charter
Uninitialized variables	Set/use analysis

The errors (misinterpretation of language constructs and hardware operation, self-modification of instructions, and loss of index or state data) are detected by review and, of course, by tests. Overflow, scaling and approximation, mishandling of singular and critical values, errors in computing equations, and incorrect values for constants are best detected by simulation and tests.

Code not Reviewed, Incomplete Testing of Modules--Contrary to the overwhelming experience that independent review of code pays large dividends, most flight control code is not reviewed. It is left to the coder to check out his work and pronounce it ready. This, and inadequate testing, allows a lot of module coding errors to slip into the integration stage.

Typographical Errors--Many typographical errors will be caught by the static analyzer. Spelling errors introduce new variables which are detected by set/use analysis. Errors in signs are more difficult to find. A simulation will show that the calculation is not correct.

Incorrect Implementation of Design Specifications--The methodology can contribute here. The modules for which transition tables are used as specifications or for which output assertions may be written can be easily verified.

Poor Documentation--The recurring problem of poor documentation has been covered sufficiently.

RELIABILITY AND CONFIDENCE

With a combination of techniques and sufficient care, it is possible to verify with certainty all of the particular functions for a triply-redundant flight control system. These functions are elementary when compared to those in more general software systems. This basic simplicity may be exploited to demonstrate that each software function is performing precisely as it is intended.

How then can we be sure there are no strange demons and traps erroneously introduced into the system functions? Can a diabolical programmer produce a construction which calls a fatal flaw under subtle circumstances? Why is it certain that these types of errors cannot slip past our verification procedures? A detailed analysis through all of the code for the generic system shows that there need be no singular points and dangerous structures in the program. Reasonable care in the structure, design, and documentation is all that is required for the independent verifier to check through each function.

The previous discussion applies only to individual functions. The global consistency of the system cannot be asserted so categorically. The case which may be cited is the possibility of a single-point failure due to conflicting interpretations of faults by the numerous self-tests and cross-channel monitors. The analysis of failure mode effects is the most difficult part of the validation work. This is actually part of the system development. However, errors of this nature are almost always ascribed to the software. They must be fixed in the software.

In these problems, the tools and techniques have much less to offer. It is doubtful that the formal methodologies will be of much use. The approach advocated is to limit the design options so that each level may be theoretically verified and validated as the design progresses to more detailed levels of implementation. In the SIFT system, for example, cross-channel monitoring is the only mechanism used for failure detection and hence, there are no possible conflicting self-tests. This may be a high price to pay for confidence bought by formal analysis. However, it is certain that at the function level attention to verification improves the design.

SECTION V

IMPACT ON SPECIFICATIONS

Specification impact can be viewed from the standpoint of the general FCS specification, MIL-F-9490D, or a particular specification to satisfy a set of system requirements. Additionally, the impact can be viewed on the computer program level as well as on the system level. The intent here is to assess the impact of the V&V methodology from all of these viewpoints. Ultimately, however, the major emphasis is on particular system-level specifications.

Following a review of key specification-related documents, the V&V requirements are incorporated into the quality assurance sections of a hypothetical system specification and a related computer program specification. Various aspects of fulfilling these requirements are then assessed as further background to the recommendations on specification impact.

SPECIFICATION-RELATED DOCUMENTS

The impact of the V&V methodology should be assessed with regard to the general FCS specification, MIL-F-9490D, and with regard to the acquisition specifications for particular digital systems. MIL-F-9490D impact, moreover, may be more extensive in terms of the amount of text involved for its user guide, AFFDL-TR-74-116, than for the specification document itself. The rather general provisions affected in them become more detailed and concrete as they are developed and applied in acquisition specifications,

which are normally prepared in compliance with MIL-STD-483 and MIL-STD-490. Certain parts of MIL-STD-483 are especially relevant to the practical and successful utilization of the methodology.

Basically, verification and validation are concerned with assuring compliance with appropriate specifications as illustrated in Figure 1. This involves a statement of the requirements whose fulfillment is to be confirmed and of the methods and procedures whereby this can be accomplished. The design requirements encompass the functional and performance features which the system is to incorporate; these are complemented by the corresponding assurance requirements to substantiate their correct implementation. Some latitude then exists to select or adapt assurance methods to provide this substantiation.

This latitude, moreover, presents an ideal opportunity to invoke and apply an integrated V&V methodology, such as one of those developed under this study. The essential concern is to tailor the methodology to the needs of a particular project and to incorporate the methodology into the earliest stages of planning. This is appropriately done under the FCS development plan required by MIL-F-9490D and, with respect to software, under the computer program development plan (CPDP) defined by Data Item Description DI-S-30567A. Both of these are subject to approval by the procuring activity, so the Air Force now has some leverage to influence the V&V methods selection.

Figure 7 depicts the more important effects which these respective documents have on the DFCS development process. The intent is to show how and at what points these documents come into play, not to depict the entire process. More particularly, the timely definition and appropriate adaptation of the

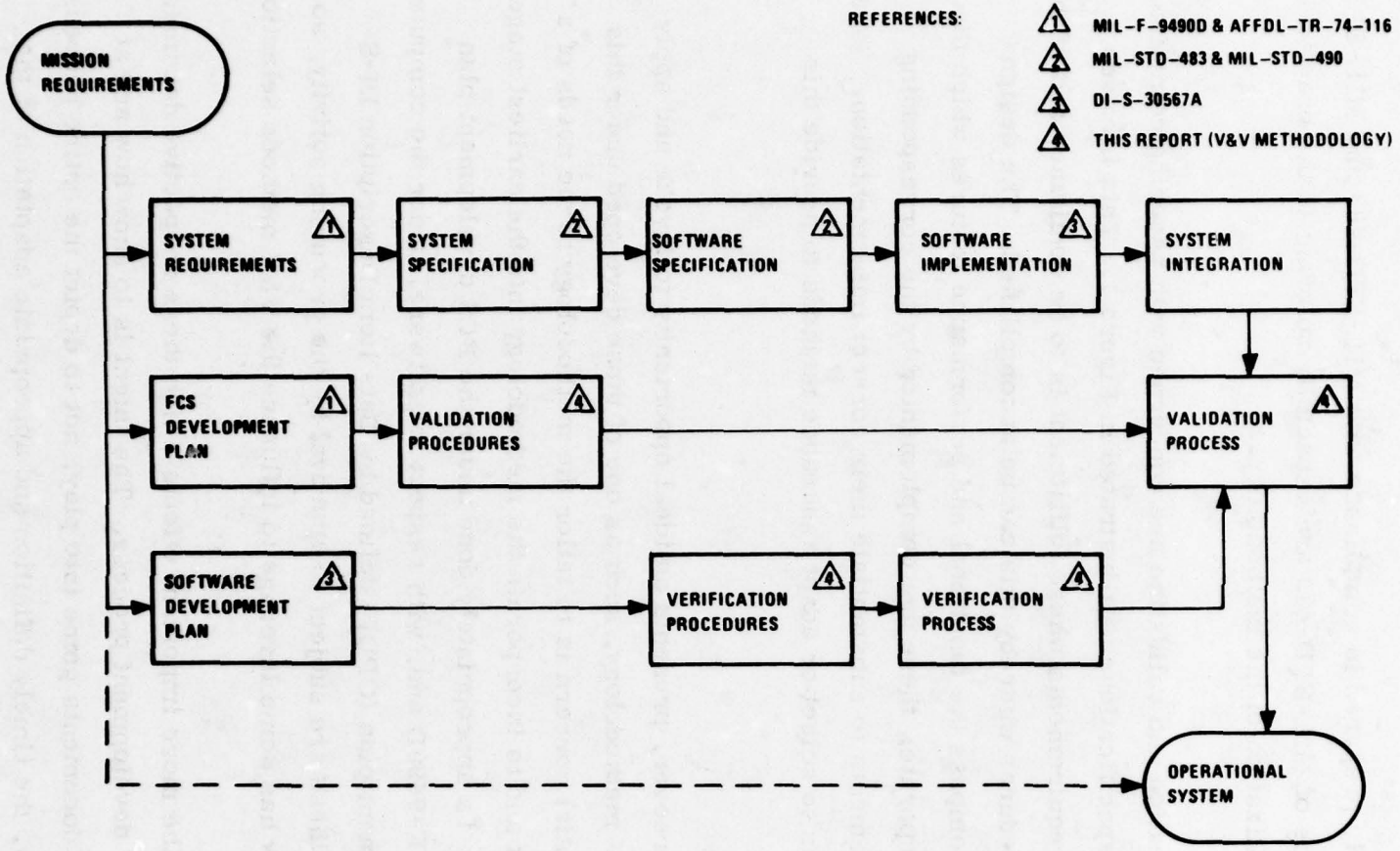


Figure 7. Impact of Specification-Related Documents

V&V methodology constitute the key elements in pursuing an effective development cycle. Planning which acknowledges and incorporates the recommendations of this study, in consonance with the referenced Air Force documents, contributes significantly to the quality and cost effectiveness of DFCS software as well as to that of the overall system itself.

To amplify their specific roles and contributions, the salient details of each of the five documents identified above are presented in the following sections. Their actual or potential applicability to the DFCS engineering process and their relevance to the V&V methodology are then developed. Particular attention is devoted to the flexibility of each in fulfilling differing needs of various DFCS projects. This discussion then serves as the basis for formulating and substantiating study recommendations relative to specification requirements and AFFDL-TR-74-116.

Critique of MIL-F-9490D

Table 17 summarizes and classifies the potential impact of this study on MIL-F-9490D. The specification sections ranked "1" in the urgency column require changes to accommodate the essential results of this study. A ranking of "2" indicates a likely candidate for changes, and a "3" denotes a low-priority change or one emanating from digital technology outside the scope of this study.

Although the potential impact for each section cited is noted in Table 17, further details may be needed to clarify the exact intent. Accordingly, each MIL-F-9490D section ranked "1" is discussed individually below.

TABLE 17.

STUDY IMPACT ON MIL-F-9490D/AFFDL-TR-74-116

1 - High Priority
2 - Intermediate Priority
3 - Low Priority

MIL-F-9490D Section	EFFECTIVENESS -9490D	EFFECTIVENESS TR-74-116	ASPECT Gen. S/W	ASPECT V & V	ASPECT Other	POTENTIAL IMPACT	URGENCY
1.1		X			X	Note explicitly that software is a component of the system	3
1.2.1.1	X				X	Include software among types of components	3
2.1	X				X	Additional specifications, e.g., MIL-S-52779	2
2.1	X				X	Additional standards, e.g., MIL-STD-483, -490	2
2.2	X				X	DOD Directive 5000.31 on Higher-Order Languages	3
3.1.1		X			X	Selectable command mode switching must not trip comparators	3
3.1.1		X			X	Discrete signal granularity must not contribute to limit cycles	3
3.1.1		X			X	Sidearm controller use must be reconciled to stick forces	3
3.1.2		X			X	Damping factors, residual oscillations, and standoff errors must meet specified values despite discretization	3
3.1.2.10.1	X	X			X	Numerical noise must be considered among non-ideal effects	3
3.1.3.1		X			X	Also consider software implemented redundancy	3
3.1.3.2	X	X	X			Also consider generic software error in replicated channels	2
3.1.3.3.3	X	X			X	Preclude comparator faulting during mode switching	2
3.1.6.1		X			X	Stability margins must be set despite nonminimum phase system	3
3.1.3.6.2		X			X	Consider aliasing, granularity, asynchronous sampling, etc.	3
3.1.3.8		X			X	Discretization must be compatible with osillatory limits	3

TABLE 17.

STUDY IMPACT ON MIL-F-9490D/AFFDL-TR-74-116 (continued)

MIL-F-9490D Section	EFFECTIVENESS -9490D	EFFECTIVENESS TR-74-116	ASPECT Gen. S/W	ASPECT V & V	Other	POTENTIAL IMPACT	URGENCY
3.1.3.9	X	X	X			Also consider latent software errors	2
3.1.3.9	X	X			X	Also consider CPU tests and hardware monitors	3
3.1.3.9.1	X	X			X	Also consider coverage issues	3
3.1.3.9.2	X	X			X	Also consider self-monitored channel	3
3.1.4.2		X		X		Validation (not verification) in simulated user environment	2
3.1.5.3.1		X			X	Fly-by-wire likely to be MFCS, rather than AFCS	3
3.1.6 & 3.1.7	X		X			Material failures exclude software failures per se	2
3.1.6 & 3.1.7		X	X			Software reliability is not addressed	2
3.1.9.5	X	X	X			Need to address software maintenance safeguards	1
3.1.10	X					Maintenance software capability noteworthy	3
3.2.3.3	X	X			X	Relevance to multiaxis MUX buses should be clarified	3
3.2.3.3.2	X	X			X	Reference MIL-STD 1553B and acknowledge revisions	3
3.2.4.1.3.1	X	X			X	Fiberoptics is hardly nonconventional nowadays	3
3.2.4.3.2	X		X			Change "resident and bulk" to "program and workspace"	3
3.2.4.3.2	X		X			Program memory growth need only be potential availability	3
3.2.4.3.2.3	X	X			X	Clarify "non-programmable" computer intent	3
3.2.4.3.2.3	X	X		X		Describe V&V methodology impact on maintenance relative to support software and documentation	1
3.2.4.3.2.3		X			X	Apparently "on computer" should be "or compiler"	3
3.2.4.3.2.3		X	X			Augment specified documentation list	1

TABLE 17.

STUDY IMPACT ON MIL-F-9490D/AFFDL-TR-74-116 (continued)

MIL-F-9490D Section	EFFECTIVENESS -9490D	EFFECTIVENESS TR-74-116	ASPECT Gen. S/W	ASPECT V & V	Other	POTENTIAL IMPACT	URGENCY
3.2.7	X		X			Consider software type components sections on design	3
3.2.7.1.2	X					Consider impact software changes within box	3
3.2.8.2	X		X			Consider software design process sections	2
4.1.1	X			X		"Validation", rather than "verification"	2
4.1.1	X	X		X		Methodology may determine selected methods	1
4.1.1		X		X		"Judgment" relating to level of methodology selected	1
4.1.1.1	X			X		Analysis may be preferable or mandatory for some aspects	1
4.1.1.1	X			X		Certain software analyses should be listed	1
4.1.1.2	X	X		X		Include walkthrough or code inspections	1
4.1.1.3	X			X		Emphasis on testing seems too strong	1
4.1.1.3		X		X		Integrated V&V should be acknowledged and fostered	1
4.2	X			X		Define limitations and coverage for software analyses	2
4.2.1	X			X		Investigate discretization effects, e.g. display-command coarseness	3
4.3.1.2	X				X	No acceptance tests for software per MIL-STD-483	3
4.3.1.3	X			X		Instrumentation of software is of possible interest	3
4.3.2	X			X		Software V&V sections should be added	2
4.3.2.3	X			X		Flight-critical software needs special consideration	2
4.3.4	X			X		"Validation," not verification	3
4.4	X	X	X			Need some blanket section on software documentation	2
4.4.1	X		X			Explicitly call out computer program development plan (CPDP)	1

TABLE 17.

STUDY IMPACT ON MIL-F-9490D/AFFDL-TR-74-116 (concluded)

MIL-F-9490D Section	EFFECTIVENESS -9490D	EFFECTIVENESS TR-74-116	ASPECT Gen. S/W	ASPECT V & V	Other	POTENTIAL IMPACT	URGENCY
4.4.1	X	X		X		Outline or reference integrated V&V methodology as part of the CPDP	1
4.4.1a & c	X	X		X		"Validation or verification" instead of just verification	2
4.4.1a	X				X	May wish to acknowledge MIL-STD-1521A and AFR-800-14 considerations	3
4.4.1c	X	X		X		Allude to integrated DFCS V&V methodology	2
4.4.1d	X		X			Note software "ilities" considerations	2
4.4.1e	X			X		Note impact of V&V methodology	2
4.4.1g	X		X			Note inflight software change provisions	3
4.4.1		X		X		Revise or augment *Table IIID for DFCS*	3
4.4.2	X				X	May wish to reference MIL-STD-483 and -490	3
4.4.2	X		X			Call out software specifications for DFCS	3
4.4.3.1	X		X			Add section on software analyses and documentation	1
4.4.3.1e	X				X	Note discretization consequences	3
4.4.3.1f	X				X	Note software impact on "ilities"	3
4.4.3.2	X			X		Report results of walkthroughs	3
4.4.3.3	X			X		Report V&V test results	1

Section 3.1.9.5--The V&V methodology described in this study is development-oriented, but the principles and methods on which it is based also foster improved software maintenance practices. Hence, the methodology is readily applicable over the life cycle of the flight software, and appropriate instructions regarding its maintenance utility should be stated. This might be done under this section or else in a new one under Section 3.1.9.

Section 3.2.4.3.2.3--The software support package includes automated tools, specifications, V&V procedures, software analyses, program documentation, etc. To assure completeness of this package, it may be desirable to state this section in more forceful and specific terms. If the V&V methodology is to become generally applicable, this section should include specific elements which facilitate software maintenance on such a basis. Also, it is proper to deliniate those support items which contribute to the modification of software and those which aid in assuring the acceptability of the changes made. The methodology, moreover, could be cited as a means to foster the proper, integrated use of the software support package.

Section 4.1.1--In the case of embedded software, specific methods are, in general, required by the V&V methodology. This should be made clear at this point.

Section 4.1.1.1--One of the basic motivations for this study has been to reduce the emphasis on testing of flight software and to devise a V&V methodology oriented largely towards analytical techniques and automated tools. Not only can this approach produce cost and schedule benefits, but better quality software as well. As the methodology indicates, however, testing, analysis, and tools are not disjoint elements of the V&V process.

Their integrated use is the key to an optimized methodology. This section, then, should reflect the admissibility of and need to perform certain software analyses in addition to some testing.

Section 4.1.1.2--Code inspections or walkthroughs have become vital elements in the process of producing quality software. The walkthroughs, moreover, apply to the verification of specifications, test documents, and software design as well as to code listings. This is reflected in the V&V methodology, and this section should be expanded or augmented to underscore the role of software-oriented inspections.

Section 4.1.1.3--Again, the emphasis on testing is inappropriate or overstated. It needs to be modified or qualified to acknowledge the benefits and need for software analytical techniques. Possibly all of Section 4.1.1 should distinguish between the software-oriented requirements and those of a strictly systems nature. This is undesirable, however, from the standpoint of masking the role of the embedded software as an integral component of the system.

Section 4.4.1--The minimum list of elements to be included in the flight control system development plan is quite extensive, but none of these specifically address any of the aspects of digital implementation. This may well serve to maintain the generality of the stated provisions; nonetheless, the section seems to be the crucial point at which the overall integrated V&V methodology needs to be called out. Possibly this could be done in an additional paragraph which would apply only to DFCS. It might also be desirable to reference the CPDP as an adjunct of the FCS development plan.

Section 4.4.3.1--Again, the minimum list of elements to be acknowledged is rather general. It does seem essential, however, to include specific provisions which call for certain V&V methodology results in the FCS analysis report. These would include software analyses, documentation, backup data, etc., along with descriptions of their nature, origins, and significance.

Section 4.4.3.3--Similarly, the FCS test report should be required to reflect V&V methodology-oriented test data for digital system mechanizations. The significance and completeness of these data should be stipulated as well, along with test confirmation of prior software analysis.

Specific recommendations for text modifications to MIL-F-9490D are offered later in this report. The point here is to note the nature and basis for the more urgent recommendations. Of somewhat less compelling significance are the intermediate priority candidates for specification revisions. These are presented in Table 18.

Critique of AFFDL-TR-74-116

Some extent of modification to the MIL-F-9490D User Guide is desirable, regardless of whether a V&V methodology guidebook is prepared. The changes discussed below are all ranked "1" in Table 17, so they are considered important.

TABLE 18.

IMPACT OF INTERMEDIATE PRIORITY CONSIDERATIONS ON MIL-F-9490D

MIL-F-9490D SECTION	POTENTIAL IMPACT	COMMENTS
2.1	Citation of MIL-S-52779	Software quality assurance activity for flight-critical functions
2.1	Citation of MIL-STD-483, MIL-STD-490	System and computer program specification content/format
3.1.3.2	Proscription of Generic Software Error	Coincident channel shutdown unacceptable where redundancy is needed
3.1.3.3.3	Compatibility of Software Mode Logic	Deadlocks, nuisance trips, and other anomalies inadmissable, especially during multichannel operation
3.1.3.9	Tolerance of Software Errors	Incidence and effects of software errors must be bounded; latent errors of particular concern
3.1.6	Effect of Software on Mission Reliability	Software as well as material failures should be acknowledged
3.1.7	Effect of Software on Flight Safety	Software as well as material failures should be acknowledged
3.2.8.2	Process of Developing Software	Possible point to introduce and define software development process
4.1.1	Distinction of V&V Methods	Validation usually applies to system or the embedded software, verification to software in isolation
4.2	Scope of Software Analysis	V&V methodology's analytical techniques should be addressed
4.3.2	Software V&V Testing	Explicit sections on software V&V testing
4.3.2.3	Safety-of-Flight Tests	For flight-critical DFCS, certain special assurances should be considered prior to first flight
4.4	Documentation	Special section on software documentation
4.4.1	FCS Development Plan	See Table 17

Section 3.1.9.5--The point about "increasing complexity" and its relevance to "maintenance error" applies to DFCS software. This should be amplified along with the V&V methodology's use in software maintenance.

Section 3.2.4.3.2.3--Here the contents of the software support package should be detailed to reflect the provisions of the V&V methodology. Certain design- or development-oriented items would not be included, but V&V tools need to assure the quality of modified software would be.

Section 4.1.1--The "judgment for selecting verification methods" is restricted somewhat by the use of a methodology, even though various levels of depth are permissable. It may be necessary to state this and to leave selection of the level of methodology up to the discretion of the contractor.

Section 4.1.1.1--The need for and preferability of certain software analytical techniques should be dealt with in some detail. Also, the complementarity of analysis and testing should be discussed.

Section 4.1.1.2--Review and inspection of flight software and its related documentation should be described and related to the overall V&V process.

Section 4.1.1.3--"Verification by test" is not "the preferred method," even though some amount of testing is clearly necessary. Also, the need to obtain test results faster and in a more purposeful manner should be discussed.

Section 4.4.1--The recommended methodology should clarify and strengthen "the verification plan.....whereby the contractor interfaces with the procuring agency." This plan should be described in some detail with regard to its

treatment of embedded software, and the CPDP should be accorded specific discussion.

Critique of MIL-STD-483

The purpose of this MIL-STD, "Configuration Management Practices for Systems, Equipment, Munitions, and Computer Programs," is to establish uniform configuration management practices that can be tailored to all USAF systems and configuration items, including computer programs. MIL-STDs-483 and -490 are complementary in stating the requirements for the design and product specifications of computer programs. Appendix VI of MIL-STD-483 presents the requirements for the content of a Computer Program Configuration Item (CPCI) detailed specification. Part I, Performance and Design Requirements, and Part II, Product Configuration and Detailed Specification, together make up the detailed specification.

Part I: CPCI--Part I CPCI requirements are summarized in Table 19, which relates these requirements to paragraphs of the Part I document. The technical content of Part I is a presentation of the characteristics which the CPCI is to possess and the assurance activities which will be used in confirming that the CPCI possesses those characteristics. This is the problem statement for the software developers; it is not a part of the program design. It should not, in general, restrict the program structure which will ultimately be selected and, theoretically, should not reflect this structure. In practice, however, an orderly statement of the functional requirements will often suggest an appropriate structure or certain aspects of it.

TABLE 19.

MAJOR REQUIREMENTS FOR PART I OF CPCI DETAILED SPECIFICATIONS CONTAINED IN MIL-STD-483

SECTION OR PARAGRAPH	DEVELOPMENT SPECIFICATION
1.	Scope
1.1	Identification
1.2	Functional Summary
2.	Applicable Documents
3.	Requirements
3.1	Computer Program Definition • Relationship of CPCI to other equipment/computer programs
3.1.1	Interface Requirements • Computer requirements as minimum
3.1.1.1	Interface Block Diagram or Equivalent • Graphic portrayal
3.1.1.2	Detailed Interface Definition • Quantitative terms with tolerances, where applicable, to level of detail permitting CPCI design • I/O in terms of data rate formats etc.
3.2	Detailed Functional Requirements • Quantitative terms with tolerance • Functional block diagram or equivalent graphic portrayal • Functional operation of CPCI • Not intended to restrict design

TABLE 19.

MAJOR REQUIREMENTS FOR PART I OF CPCI DETAILED SPECIFICATIONS CONTAINED IN MIL-STD-483 (continued)

SECTION OR PARAGRAPH	DEVELOPMENT SPECIFICATION
3.2.X	Function X • One paragraph for each identified CPCI function • Descriptive material • Relationship to other functions
3.2.X.1	Inputs • Source and type • Units of measure • Limits/ranges • Accuracy/precision required • Frequency of input
3.2.X.2	Processing • Text and math description of fcn • Intent: specific I/O parameters and processing • Approach: textual description of each mathematical operation, with accuracy, sequence, and timing of event; restrictions and limitations
3.2.X.3	Outputs Destination and types of output • Units of measure • Accuracy/precision requirements • Frequency
3.2.N	Special Requirements May include but not limited to: • Programming standards

TABLE 19.

MAJOR REQUIREMENTS FOR PART I OF CPCI DETAILED SPECIFICATIONS CONTAINED IN MIL-STD-483 (concluded)

SECTION OR PARAGRAPH	DEVELOPMENT SPECIFICATION
3.2.N.1	Human Performance
4.	Quality Assurance Provisions • Requirements for formal verification • Designates verification requirements and methods • Methods may include: inspection review of analytical data demo tests review of test data • Permit identification of verification requirement with specification requirement • Shall not include detail test planning documentation

Since the methodologies proposed do not include the performance and design requirements, there is no conflict between them and the Part I specification. Thus, comments about this portion of MIL-STD-483 are not made in the context of impact on the methodologies, but in the context of the status of software engineering in general.

Paragraph 3.1.1 and its subparagraphs are to present a detailed statement of the interfaces of the CPCI and all equipment and software external to the CPCI. The information required is quite complete and MIL-STD permits the information to be presented in the form most appropriate for the particular instance.

Section 3.2 presents detailed functional requirements in quantitative terms. For each CPCI function, the relation between it and other CPCI functions must be shown, as well as the relation to other computer programs and hardware equipment. The primary requirement is to call out the software functions required by each DFCS operating mode and state the general requirements for synchronization and system control. This requirement may be met by HIPO charts, although these may be somewhat different than the HIPO charts which would be developed later to show the computer program structure. HIPO charts, however, are well suited to the task of functional description in that the input, processing required, and use of the output are all shown.

For each function, the source of input is to be specified, along with the units of measure, limits or ranges, accuracy/precision requirements, and frequency of input information arrival. The complete specification of inputs in this detail may require supplementary notes to augment the HIPO notation or a diagram showing the flow of data between functions.

The statement of requirements may also be expressed in a more formal way by means of a language such as PSL (Problem Statement Language), possibly with some amplifying and supplementary material. Within the next several years, it may be appropriate for MIL-STD-483 to be revised to require the use of such a language. Such a revision is not currently appropriate and should not precede the accumulation of substantial experience with such languages. This experience would have to confirm that the language(s) approved would enhance overall utility of the document and/or significantly reduce the cost of preparing it.

Special Requirements, Paragraph 3.2.n (where n denotes the nth function) is to contain requirements which are distinguishable from the performance requirements of the CPCI. These may include programming standards and special features to facilitate testing. We expect that this paragraph would state the programming language to be used and provide a statement of the methodology. This paragraph does not then deal with methodology requirements but with a description of the development process to be used. Consequently, there is no conflict with the methodology itself.

Paragraph 3.2.n.1 deals with human performance. Since DFCS software is embedded, human performance considerations will normally be stated as system requirements. These will be translated into hardware and software requirements which will not be particularly identifiable as human performance requirements. Hence this paragraph is not expected to relate to the methodology either positively or negatively.

The thrust of Section 4 is to present the means of verifying each performance or design requirement specified in Section 3. Acceptable verification methods

are inspection of the CPCI, review of analytical data, demonstration tests, and review of test data. The techniques and tools in the proposed methodologies appear to include a broader range than that identified in MIL-STD-483. However, such formalized methods as symbolic evaluation and set-use checking are very definitive inspection techniques; consequently, these are allowable under a liberal interpretation of the MIL-STD. Nonetheless, it is deemed preferable to revise the MIL-STD to specifically acknowledge a broader range of verification methods.

Appendix VIII of MIL-STD-483 presents amplified instructions for the preparation of CPCI specification change notices and other activities to be performed in maintaining the currency of the CPCI specification. Thus, the content of this appendix is largely procedural and has minimal impact on the proposed methodologies.

Part II CPCI--Part II of the CPCI specification contains a comprehensive description of the computer program developed in response to the Development Specification. Part II is intended to provide all of the information necessary for maintenance programmers to correct any deficiencies in the original CPCI which escape the V&V process and to modify the program as necessary to extend its useful life. While the requirements stated in MIL-STD-483 are extensive, they are not excessive for a specification which is intended to apply to a broad range of software, including very large CPCIs. The intent of these requirements does not conflict with the intent of the proposed methodologies, but the implementation details do create some conflicts. The following discussion addresses the requirements relative to the methodologies.

Sections 1., Scope, and 2., Applicable Documents, require some specification text which is neither relevant to nor in conflict with the methodologies.

Section 3., Requirements, is an extensive section, as can be seen from Table 20. The CPCI top-level configuration can be shown by the top-level HIPO charts or by the top-level descriptions in formal specification language. Details of the CPCI configuration are shown in the various paragraphs of Section 3. Since the flight software CPCI consists of a single computer program component (CPC), these top-level presentations also satisfy the requirement for CPCI structure and functions.

Although the requirement to describe the data base is more applicable to data processing programs, the data base can be considered to be described in Paragraph 3.2.1.3. A reference to this paragraph should suffice.

The requirement to describe all CPCs is inherently satisifed by the description of the CPCI in the case of DFCS software.

Symbolic name assignment conventions and register usage conventions may require some supplementary material not included in the methodologies, but this poses no problem.

Following the overview material in the lead-in portions of Section 3, the presentation of material gets more detailed. Paragraph 3.1 requires that the structure and functions of the CPCI be presented. This may be done here and referenced in the lead-in material, or vice-versa. Paragraph 3.1 also requires that the allocation of functions to CPCIs be presented. This does not apply in the case of a CPCI consisting of a single CPC.

TABLE 20.

MAJOR REQUIREMENTS FOR PART II OF CPCI DETAILED SPECIFICATIONS CONTAINED IN MIL-STD-483

SECTION OR PARAGRAPH	PRODUCT SPECIFICATION
1.	Scope
2.	Applicable Documents
3.	Requirements • CPCI detailed configuration • CPCI structure and functions • Data base • Description of all CPCs • Symbolic name assignment conventions • Register usage conventions
3.1	Functional Allocation Description • Structure and functions of the CPCI • Allocation of functions to CPCs
3.2	CPC Descriptions
3.2.1	CPC 1
3.2.1.1	CPC Description • Program logic • Data flow • Equations • Algorithms • Timing and accuracy • Special conditions • Words, figures, equations, references to flow charts of 3.2.1.2

TABLE 20.

MAJOR REQUIREMENTS FOR PART II OF CPCI DETAILED SPECIFICATIONS CONTAINED IN MIL-STD-483 (continued)

SECTION OR PARAGRAPH	PRODUCT SPECIFICATION
3.2.1.2	CPC Flow Chart • Graphic portrayal of CPC operations • (Series of) flow chart(s) • Minimal detail consistent with understanding operation and data flow
3.2.1.3	CPC Interfaces • Set-use matrix • Table, item, buffer descriptions • Input/output formats • May be by reference to other Part II paragraphs
3.2.1.4	CPC Data Organization • All data items and tables unique to CPC • Memory available for temporary storage • Internally defined symbols and constants
3.2.1.5	CPC Limitations • Timing requirements • Limitations of algorithms and formulas • Input/output data limits • Error sensing and error checks
3.2.1.6	CPC Listing • Relatable to flow diagrams

TABLE 20.

MAJOR REQUIREMENTS FOR PART II OF CPCI DETAILED SPECIFICATIONS CONTAINED IN MIL-STD-483 (continued)

SECTION OR PARAGRAPH	PRODUCT SPECIFICATION
3.3	Storage Allocation • Graphic portrayal • Data base, computer program • Pertinent timing, sequencing, and equipment constraints which influence allocation • If specification and graphic portrayal infeasible, then allocation algorithms
3.3.1	Data Base Characteristics • File description • Table description • Item description • Graphic table description • List of constants
3.4	Computer Program Functional Flow Diagram • Information flow • Control flow • Modes distinguishable
3.4.1	Program Interrupts • Source, purpose, type, required response to each interrupt
3.4.2	Subprogram Reference Logic • Timing and sequencing • Priority assignments

TABLE 20.

MAJOR REQUIREMENTS FOR PART II OF CPCI DETAILED SPECIFICATIONS CONTAINED IN MIL-STD-483 (concluded)

SECTION OR PARAGRAPH	PRODUCT SPECIFICATION
3.4.3	Special Control Features
4.	Quality Assurance • Test plans and procedures used in CPCI qualification, by reference • Test requirements for CPCI duplication by specification or reference
4.1	Test Plan/Procedure Cross-Reference Index • CPCI functions to test activity • Special tools
4.2	Other Quality Assurance Provisions • Requirements, methods, and procedures related to preparation and duplication
5.	Preparation for Delivery
5.1	Preservation and Packaging
5.2	Markings
6.	Notes

Paragraph 3.2 and its subparagraphs address the details of the program design. Paragraph 3.2.1.1 requires that program logic and data flow be presented. The HIPO charts satisfy this requirement, as does the formal specification language. However, a graphic depiction of data flow would no doubt be desirable to augment a specification language presentation. The requirement to present the equations and algorithms used could be satisifed by either the HIPO or specification language approach if properly implemented. Similarly, timing, accuracy, and special conditions can be expressed in either medium. The freedom to use words, figures, equations, and references to Paragraph 3.2.1.2 permits a helpful latitude in meeting these requirements.

Paragraph 3.2.1.2 calls for a graphic portrayal of the CPC operation through a (series of) flow chart(s). This requirement is not inherently satisfied by the proposed methodologies. Revision of the MIL-STD to recognize available documentation forms other than flow charts seems advisable.

Paragraph 3.2.1.3 covers CPC interfaces. In the case of DFCS software, it is appropriate to include the set-use matrix called for. This should probably be included in Paragraph 3.2.1.1 and referenced here. Similarly, the table, item, and buffer descriptions may be covered in 3.2.1.1 and referenced here. Input/output formats should also be covered in 3.2.1.1, under the proposed methodologies. This paragraph, however, offers an opportunity to present a consolidated description of the interfaces which appear appropriate for DFCS programs with extensive mission applications requirements.

Paragraph 3.2.1.4, CPC Data Organization, will normally be covered in preceding paragraphs in the case of DFCS software.

Paragraph 3.2.1.5 does not conflict with the methodologies proposed. Timing requirements and input/output data limits will normally be included in 3.2.1.1 and reference should be sufficient here. Limitations of algorithms and formulas used can be presented here without conflicting with the methodology. Error sensing and error checking are important functions of the DFCS software/system design. Thus, these will normally be covered in 3.2.1.1.

When Paragraphs 3.2.1.1 through 3.2.1.5 are considered as a group, there are two areas where conflict with the methodologies is evident. One is the previously mentioned requirement for flow charts to be used. The second is that the proposed methodologies tend to present the required information in a consolidated form in 3.2.1.1 rather than distributing it through the several paragraphs. This poses no serious problem if liberal use of references is accepted by the contracting office.

Paragraph 3.2.1.6 calls for a listing of the CPC which is relatable to the flow diagrams of Paragraph 3.4. This relatability is fostered by the proposed methodologies, but relatability to HIPO charts or the formal design language should be allowed if these are being used.

Paragraph 3.3 covers assessment of resource allocation, such as storage and execution time. Without detailing the particular requirements, it may be noted that they are covered in 3.2 (for example, timing constraints) or may be responded to here (for example, list of constants). No conflict with the methodologies is foreseen.

Paragraph 3.4. requires information flow and control flow diagrams. These constitute a redundant presentation of information, under the proposed methodologies, and could be waived without decreasing the usefulness of the document

Program interrupts, required to be listed in Paragraph 3.4.1, are usually an integral part of program control in DFCS software. Consequently, they may be listed here with reference to 3.2.1.1 for source, purpose, type, and required response.

Subprogram reference logic is usually an integral part of DFCS software control structure and information flow. Consequently, the information called for in Paragraph 3.4.2 is normally covered in 3.2.

Section 4 covers quality assurance aspects of program preparation. Testing is the only method mentioned which relates to quality assurance during design and programming. There is certainly no conflict here with the proposed methodologies, but adoption of the recommendations for Part I to cover more than testing would make a corresponding change here appropriate. The quality assurance requirements for replication of the program are also covered in this section. Since the software copies will normally be made in programmable read-only memory, a program read-back and comparison should be adequate. This is outside the scope of this study, however.

The topics in Section 5, Preparation for Delivery, are not directly germane to the subject of this report, the evaluation of software methods. They are important, however, in preserving the quality of software production copies throughout the system life cycle.

Section 6, Notes, is not affected by the methodology.

Critique of MIL-STD-490

MIL-STD-490, Specification Practices, establishes the format and content of specification documents of a number of types. Those types pertaining to computer programs are B5, Development Specification, and C5, Product Specification. These are, respectively, the Part I and Part II specifications discussed in MIL-STD-483. The requirements presented in MIL-STD-490 are compatible with but less detailed than the presentation in MIL-STD-483, which means a detailed discussion of MIL-STD-490 would be redundant. Consequently, the discussion here deals only with specific areas where revision is suitable.

Appendix XIII requires that a functional flow diagram be included in Paragraph 3.4 of the C5 specification. The wording is not particularly constraining, and acknowledgement that the requirement can be satisifed with HIPO charts would be sufficient.

Critique of DI-S-30567A Computer Program Development Plan

This data item, usually referred to by the acronym CPDP, is a document intended to describe the detailed plan for management and development of computer programs and associated documentation. References to other documents may be used to reduce duplication of information. CPDP contains at least 18 categories of information, identified here as 10.1.a through 10.1.r.

As a composite, the information contained in or referenced by the CPDP is comprehensive. It should be noted that essentially all of the CPDP requirements may be satisfied by reference to other documents. Therefore, the

CPDP document itself serves more as an index than as a working document. This makes the CPDP preparation less burdensome than it might initially appear, but only if the management and technical issues have been meaningfully addressed in other contexts.

For computer programs developed under contract, most of the topics addressed in the CPDP must be addressed in the contractor's proposal in order to show a sound approach to meeting program requirements. Consequently, at least the framework of the CPDP material would be generated pre-contractually. In some programs, the contracting agency may require a CPDP as part of the proposal. In such cases the material would be developed to a substantial level of completeness and detail pre-contractually.

A synopsis of each of the 18 topics follows. Comments are included where DFCS software characteristics are relevant to the topic, and the CPDP is assumed not included in the proposal.

10.1.a. Requirements Assessment Summary--This section is to summarize the contractor's understanding of the software requirements and related hardware and system requirements, with emphasis on high risk and uncertainty issues. In the case of DFCS software, the contractor has generally explored these aspects in considerable detail prior to proposal preparation and included most of the information in the proposal.

10.1.b Project Objectives--The parts of this section pertinent to DFCS software include budget goals for timing and memory, with margins for growth. These tend to be straightforward for DFCS, since there is a single CPCI, and often will be in the proposal. The relative importance of software

characteristics such as reliability, maintainability, and testability is also required. These will tend to be similar for most DFCS programs, with reliability, testability, and efficiency most important and other attributes such as portability being of relatively little concern.

10.1.c. Work Definition--This section elucidates the tasks involved in the development and delivery of the software. Most of this section will be fairly straightforward. The testing and quality assurance tasks should be emphasized for DFCS software, expanding the proposal material. Because of the nature of flight software, the test activities are straightforward but may be extensive.

10.1.e. Activity Network--The generation of an activity network for DFCS software development will be fairly straightforward from the task breakdown and schedule information. More meaningful for larger programs, the activity network could be waived for DFCS software.

10.1.f. Organization--The contractor organization in terms of groups, subcontractors, responsibilities, lines of communication, interfaces, and skill requirements is a repeat of proposal material, possibly with more detail. The other material in this section, such as relationships among contractor and subcontractors, independent verification and validation groups, and customer agencies is more meaningful, in that these relationships change from program to program. Care is warranted in preparing a clear definition of these organizational interfaces.

10.1.g. Resource Allocation--This section presents the provisions made by the contractor to assure the availability of support resources required during development of the CPCI. These include hardware, such as prototype or target computers and existing computer facilities. These resources also include software resources, such as compilers and automated tools. Normally, in DFCS development, the resources are fairly uniform from program to program, although this section will help focus attention on the adequacy of computer resources and the availability of new software resources, for example, a compiler or assembler.

10.1.h. Engineering Standards--This section presents the contractor's approach to software standards and methods for enforcing those standards. For DFCS software, these will tend to center on the fundamentals of structured programming, as practiced by the contractor. The engineering standards will also include programming language standards. The programming standards will most often be identified by reference to the contractor's standards manuals.

10.1.i. Design Assurance Techniques--This section of the CPDP is one of the most enlightening on the subject of the contractor's ability to produce the highly reliable software required for DFCS applications. The contents will show the contractor's understanding of the role of various tools and techniques throughout the software life cycle and their integration into a viable and efficacious methodology. In this section, the contractor identifies the techniques to be used in assuring that the computer program requirements will be carried forward throughout the design process so that they will be satisfied by the delivered software. This section also presents the design evaluation techniques and tools which the contractor will employ such as simulations, emulations, prototypes, and mathematical models.

10.1.j. Detailed Design, Coding, and Checkout--In this section, the contractor presents a definition of the procedures, steps, and documents associated with the detailed design, coding, checkout, review, and acceptance of individual software modules. This material relates to the contractor's management of the development of the individual program parts which will ultimately be integrated to form a CPC or CPCI.

10.1.k. Development Test and Evaluation--This material is to present the philosophy and approach for the integration and test of CPCs and CPCIs. This section has assurance implications, and the material here should be compatible with and complementary to other assurance-related sections.

10.1.l. System Test and Evaluation--This material relates to the support of CPCIs and computer resources during formal test and evaluation. This support is fairly minimal for DFCS software, since the hardware is usually also under test. The hardware support plans and the system test plan cover this area for DFCS.

10.1.m. Anomaly Control--This topic covers the detection, correction, and documentation of software anomalies and the means of maintaining configuration control. This broad area relates to the software quality assurance efforts, testing, and configuration management activity. Some of the material to be included may be redundant to the quality assurance aspects, but its inclusion here assures completeness on this topic.

10.1.n. Management Controls--The relationships between the management controls of the CPDP and other applicable management plans are identified here. This is a vague requirement, so it is difficult to know when it has

been satisfied. Means of providing management visibility of deviations from plan, such as schedule slippage, are usually far from ideal in technical development programs. Appropriate management action is often elusive, since recovery from a deviation is usually difficult or, if the deviation is large, impossible.

10.1.o. Documentation--The contractor is to present fairly comprehensive information on the documentation methods to be employed, including benefits and conditions of equivalence with other methods. The importance of clear and meaningful documentation is unquestioned; this topic emphasizes that importance.

10.1.p. Configuration Management--This topic requires that the contractor address, in a specific and definitive way, the configuration management aspects of the CPCI development. Strong configuration management is prerequisite to high software reliability.

10.1.q. Vendor/GFE Computer Resources--The thrust of the CPDP in this area is to assure that the contractor considers the adequacy of present and future hardware for execution of the software. This topic is of less relevance to DFCS software than to general application software because of the close relationship among the software, hardware, and airframe. DFCS software is not portable, and many of its functions are dependent on computer architecture, characteristics of other system components, and the airframe for which it is designed.

10.1.r. Support Resources for the Deployment Phase--The contractor is to present recommended support philosophy and resource requirements for use

after the full-scale engineering development phase. Also, the contractor is to present a plan for transfer of the computer resources including support software to appropriate Air Force agencies. This section directs attention to the very important phase of transition of the software to the customer and the support the customer will receive.

APPLICABILITY TO FLIGHT CONTROL SPECIFICATIONS

Specifications for digital flight control systems must be considered on both the system level and the software level to assess properly the impact of the V&V methodology. The emphasis must be placed primarily on the system level, towards which both MIL-F-9490D and the validation process are oriented. The intent here is, first of all, to formulate a partial DFCS specification to illustrate how the methodology might be incorporated.

Next, the interpretation and application of these specification requirements are described and critiqued. This is done in the context and phasing of the software development cycle. Lastly, ramifications of the various methodology levels are discussed, with emphasis on their impact on specifications.

DFCS Quality Assurance Provisions

Table 21 summarizes the more important assurance sections of the three primary documents which affect the form and content of DFCS specifications. MIL-F-9490D is more comprehensive and specific and hence is the dominant influence in preparing system specifications. At present there is no comparable document to define the essential details needed in a DFCS software specification, so considerable leeway exists in following the CPCI provisions of MIL-STD-483.

TABLE 21.

QUALITY ASSURANCE SPECIFICATION PROVISIONS

SECTION NO.	SYSTEM SPECIFICATION		SOFTWARE SPECIFICATION (PT. 1)
	MIL-F-9490D	MIL-STD-490	MIL-STD-483
4	Quality Assurance	Quality Assurance Provisions	Quality Assurance Provisions
4a		Reliability Testing	
4b		Development Testing	
4c		Qualification Testing	
4d		Installation Checkout	
4e		Verification	
4.1	General Requirements	General	Introduction
4.1.1	Demonstration of Compliance	Responsibility for Tests	Category I Test
4.1.1.1	Analysis		
4.1.1.2	Inspection		
4.1.1.3	Test		
4.1.2		Special Tests	Computer Program Test
4.1.3			Preliminary Qualification Tests
4.1.4			Formal Qualification Tests
4.1.5			Category II System Test
4.2	Analysis Requirements	Quality Conformance Inspections	Test Requirements
4.2.1	Piloted Simulations		
4.3	Test Requirements		
4.3.1	General Test Requirements		
4.3.2	Laboratory Tests		
4.3.3	Aircraft Ground Tests		
4.3.4	Flight Tests		
4.4	Documentation		

TABLE 21.

QUALITY ASSURANCE SPECIFICATION PROVISIONS (concluded)

SECTION NO.	SYSTEM SPECIFICATION		SOFTWARE SPECIFICATION (PT. 1)
	MIL-F-9490D	MIL-STD-490	MIL-STD-483
4.4.1	FCS Development Plan		
4.4.2	FCS Specification		
4.4.3	Design and Test Data		
4.4.3.1	FCS analysis Report		
4.4.3.2	Qualification and Inspection Report		
4.4.3.3	FCS Test Report		

Such detailed information, moreover, would not seem appropriate in MIL-F-9490D, as will be discussed under the recommendations.

The intent here is to outline the essential sections which should appear in a DFCS specification and in a DFCS flight software specification. Table 22 applies in the former instance, and Table 23 in the latter. These outlines are based on corresponding sections noted in Table 21 and the relevant elements of the V&V methodology. The following two sections illustrate hypothethical utilization of these outlines in preparing the major quality assurance sections of a system and a software specification.

System Specification V&V Requirements

The text in Appendix E simulates a partial specification of a DFCS on the system level. The generic triply redundant FBW system is assumed, so flight-critical functions are involved. Accordingly, the level of rigor of the methodology invoked indicates that a rather intensive V&V process is required. Note that the sections addressed are not treated exhaustively since the emphasis is on the requirements emanating from the V&V methodology.

CPCI Part I Specification V&V Requirements

The requirements stated in MIL-STD-483 have been used in formulating some example paragraphs of Section 4, Quality Assurance Provisions, of a CPCI Design Specification. This is presented in Appendix F, where a hypothetical DFCS is assumed, utilizing the triply-redundant computer and the non-machine methodology.

TABLE 22.

SECTION 4 SYSTEM SPECIFICATION OUTLINE

SECTION NO.	TITLE
4.	Quality Assurance
4.1	General Requirements
4.1.1	Analysis
4.1.2	Inspection
4.1.3	Testing
4.2	Analysis Requirements
4.2.1	System Analyses
4.2.2	Software Analyses
4.3	Test Requirements
4.3.1	Laboratory Tests
4.3.1.1	Software Tests
4.3.1.2	Computer Subsystem Tests
4.3.1.3	System Simulator Tests
4.3.2	Flight Tests
4.4	Documentation
4.4.1	FCS Development Plan
4.4.1.1	Computer Program Development Plan
4.4.2	Design and Test Data
4.4.2.1	FCS Analysis Report
4.4.2.2	FCS Test Report

TABLE 23.

SECTION 4 CPCI DESIGN SPECIFICATION OUTLINE

SECTION NO.	TITLE
4.	Quality Assurance
4.1	Introduction
4.1.1	Analysis
4.1.2	Inspection
4.1.3	Testing
4.2	Analysis/Inspection Requirements
4.2.1	Analysis Requirements
4.2.2	Inspection Requirements
4.3	Test Requirements
4.3.1	Category I Tests
4.3.2	Computer Programming Test and Evaluation
4.3.3	Preliminary Qualification Tests
4.3.4	Formal Qualification Tests
4.3.5	Category II System Test Program
4.4	Test Requirements
4.4.1	Host Machine Testing
4.4.2	Target Computer Testing
4.4.3	Simulator Testing
4.4.4	Flight Testing

Effect on the DFCS Development Program

Since the relationship of the integrated V&V methodology described in Section IV to the foregoing specification requirements may not be clear, explicit consideration of the subject is appropriate. Note first that even collectively the system and the computer program specifications as illustrated do not suffice to define or impose the V&V methodology. This would be dependent upon the contents of the FCS development plan and the CPDP, which in themselves require explicit directions, such as might be furnished in a guidebook.

In a typical case, the system specification might be prepared by an airframe company, and let to an avionics company. Either company might prepare the computer program specification, but the flight software would likely be implemented by the avionics company. Conceivably, a software firm might also be involved, possibly in a strictly assurance role. In any case, the contractor team would have to interface with the Air Force procuring activity in an organized and unified manner. Among other things, the Air Force would need evidence that the methodology was being applied in a complete and correct manner and that satisfactory results were being obtained.

To indicate, in part, how the acquisition program might be monitored by the Air Force, Table 24 furnishes an indication of how compliance with the specifications might pursued. The emphasis here is on the relationship between the specified requirements and the V&V methods employed in their resolution. In Table 25, corresponding documentation is also noted, since it constitues a major source of information for Air Force review purposes. These two tables might actually be considered a brief summary of the FCS development plan and the CPDP.

TABLE 24.

V&V METHODS APPLICABILITY BY PROGRAM PHASE

Computer Program Specification Section	PHASE Design	Coding	Integration
3.1.1.2 Interfaces	Data Flow Charts Units Consistency Checks Set/Use Checks Interface Format Checks Review & Walkthrough Timing Analysis	Type Consistency Checks Set/Use Checks Interface Format Checks Timing Analysis/Test Numerical Accuracy/Precision Analysis/Test Decision Tables Review & Walkthrough	Set/Use Checks Testing
3.2.1 Executive Functions	Review & Walkthrough Decision Tables Symbolic Evaluation Control Flow Diagrams Set/Use Checks Petri Nets	Review & Walkthrough Symbolic Evaluation Test Set/Use Checks	Instrumented Testing Rate Path Timing Path Sequencing Function Testing
3.2.2 Synchronization	Table of Events Symbolic Evaluation Set/Use Checks Review & Walkthrough Stimulation	Symbolic Evaluation Testing Review & Walkthrough	Testing
3.2.3 Logical	Symbolic Evaluation Decision Tables Review & Walkthrough	Symbolic Evaluation Correctness Proof Exhaustive Tests Review & Walkthroughs	Exhaustive Testing Set/Use Checks
3.2.4 Control Laws	Review Simulation	Review & Walkthrough Set/Use Checks Instrumented Testing-Underflow/Overflow Execution Time Frequency Responses	Testing Frequency Responses
3.2.5 Self-Tests	Review & Walkthrough Table of Events Simulation Decision Tables Symbolic Evaluation	Review & Walkthrough Set/Use Checks Symbolic Evaluation	Simulation Set/Use Checks

TABLE 25.

V&V DOCUMENTATION

V&V METHOD	DOCUMENTATION
Review & Walkthrough	Signed-Off Copy of Reviewed Material
Units Consistency Checks	Summary Reports
Data Flow Charts	Charts & Discussion
Interface Format Checks	Summary Reports
Decision Tables	Tables & Discussion
Set/Use Checks	Results
Type Consistency Checks	Results
Timing Test/Analysis	Results
Numerical Accuracy/Precision Analysis/Test	Results
Symbolic Evaluation	Results
Testing	Results, Test Report
Control Flow Diagram	Diagram, Discussion
Petri Nets	Diagrams, Discussion
Correctness Proof	Proof
Scaling Analysis	Analysis
Table of Events	Table, Discussion

Prospects for General Use

Because of the varying levels of formality or elaborateness, the V&V methodology inherently possesses an exceptional degree of general applicability. This can be maintained by suitably general provisions in an updated version of MIL-F-9490D and the existing provisions in MIL-STD-483. Thus, the needs of a particular program can be readily reflected in the acquisition specifications, which in turn can defer the more detailed aspects of the methodology definition to a more timely point in the program. Note that this does not preclude draft or preliminary plans during the earliest stages but does discourage premature decisions and inflexible mandating of a rigid methodology.

Ultimately, the general use and degree of acceptance of the methodology depends on favorable user experience and possibly some refinements. Both of these considerations call for a certain reasonable degree of flexibility, but this should not compromise the complementary nature of the various methods. Thus, the development plans for any given acquisition program should reflect an overall evaluation of the version of the methodology to be applied and should include an affirmation of its efficacy.

If the methodology is regarded and applied in the sense of "engineering" the flight software, its prospects for general use are good. Where software is equated with writing code or with programming in a very strict sense of the word, its adoption would be more remote. This attitude, however, is manifestly incompatible with the development of flight-critical software.

RECOMMENDATIONS

Specification Requirements Recommendations

Based on a review of DFCS specification requirements to effectuate the V&V methodology, the following recommendations are offered:

1. MIL-F-9490D should be revised to reflect general software provisions in Section 4 and reduced emphasis on testing.
2. MIL-STD-483 should be revised to be more flexible and general and to admit modern software engineering practices, for example, documentation and V&V methods, regardless of this study.
3. The FCS development plan per MIL-F-9490D and the computer program development plan (CPDP) per DI-S-30567A should be the primary documents to prescribe specifically the V&V methodology for a particular acquisition program.
4. An AFFDL Guidebook for DFCS Software Validation should be prepared, verified, and used as an explicit reference in preparing such plans. This should be done only after a successful detailed application of the methodology.
5. AFFDL-TR-74-116 should be revised in certain general regards, although the scope of this document is considered inadequate to cope with the rather detailed nature of the V&V methodology.

Detailed recommendations 1, 2, and 5 are discussed in the following sections; in-depth recommendations 3 and 4 are apt subjects for follow-on studies.

MIL-F-9490D Recommendations

The following specific changes to MIL-F-9490D are essential to the general use of the V&V methodology. Only relevant provisions of Section 4 are addressed.

- Section 4.1.1: Change "a specific method is" to "specific methods or a methodology are"
- Section 4.1.1.1: Change "hazardous" to "inadequate, hazardous"
- Section 4.1.1.1: Change "verified" to "verified at least in part"
- Section 4.1.1.2: After the second sentence, add "Where applicable, flight software specifications, documentation, and analyses shall be inspected or reviewed as part of the verification process."
- Section 4.1.1.3: Delete "Maximum"
- Section 4.4.1: Add the following paragraph: "h. Where applicable, a computer program development plan (CPDP) to define how the flight software is to be developed, documented, controlled, and verified."
- Section 4.4.3.1: Add the following paragraph: "j. Where applicable, a comprehensive system-oriented description of the flight software with regard to its design, implementation, and analytical evaluation. Representations shall be oriented toward understandability of various types, aspects, or functions of the software.
- Section 4.4.3.3: Add the following paragraph: "d. Where applicable, a summary of flight software testing over the range of conditions addressed on a system level."

AFFDL-TR-74-116 Recommendations

The following recommendations for Section 4 are submitted as essential to the system-level application of the integrated V&V methodology:

- Section 4.1.1: Add the following paragraph after the present one under DISCUSSION: "In the case of flight-critical or flight-phase critical DFCS, the V&V methods and their utilization are crucial to the point of restricting the leeway in judgment for their selection. Accordingly, an integrated V&V methodology equivalent to that described in AFFDL-TR-79-XX should be selected or adapted by the contractor."

- Section 4.1.1.1: To the list of "Requirements which will likely be demonstrated" add: "Software Analysis"
  Following the last paragraph in this section, add the following: "Software Analysis. *Software oriented toward the system-level* functioning of a DFCS will be accomplished. The significance of this analysis will be indicated, particularly as it relates to testing."

- Section 4.1.1.2: After the second paragraph under DISCUSSION, add the following paragraph: "Where digital implementations are concerned, visual inspections and walkthroughs will be performed at appropriate points during the development cycle. Various types of documentation in addition to the actual flight code can benefit from these walkthroughs, which often are done by multidisciplinary teams which bring varied perspectives to assess the emerging software. Such inspections have proven to be rather effective in the timely, low-cost elimination of many types of software problems."

- Software 4.1.1.3: In the first sentence of the DISCUSSION, change "is the preferred" to "is usually the preferred."
  After the second paragraph of the DISCUSSION, add the following paragraph: "Practical limitations on the realizability of thorough or exhaustive testing of software must be acknowledged, along with the variety of meanings attached to such phrases. Accordingly, effective use must be made of software analysis in interpreting test results."

- Section 4.4.1: After the second paragraph under the DISCUSSION, add the following paragraph: "In the case of digital implementations, the FCS development plan will include a description of software V&V procedures. These in turn will be detailed further in a computer program development plan (CPDP), which should be referenced by the system-level plan. Where flight-critical or flight-phase critical functions are involved, the V&V plans should reflect an integrated methodology equivalent to that described in AFFDL-TR-79-XX."

MIL-STD-483 Recommendations

The previous sections covering review of specification requirements include observations on the adequacy of these requirements for DFCS. It is beneficial at this point to present, in a single report section, recommendations for revision to enhance the utility of these documents in DFCS development. This MIL-STD is dated 31 December 1970, which makes it fairly old relative to software engineering and recent advances in software assurance technology. The most important recommendations concern the need to recognize alternatives to flow charts for graphic portrayal of CPCI design and to V&V methods

other than testing. Some recommendations follow for revisions to specific sections and paragraphs. These recommendations are made in the context of minimal revision to either the MIL-STD or CPCI specification outline. The outline in Table 23 suggests more extensive revision to the format of the MIL-STD; it does not, however, imply significant expansion of content.

Part I Paragraph 3.2 --This paragraph should be revised to encourage the use of HIPO charts for portrayal of CPCI function.

Part I Paragraph 4.1 Introduction--It is recommended that this paragraph be revised to encompass a software quality assurance plan instead of the current test plan and test procedures.

Part I Paragraph 4.1.1 Category I Test--It is recommended that this paragraph be modified to include assurance activities other than testing.

Part I Paragraph 4.1.2 Computer Program Test and Evaluation--It is recommended that this paragraph be modified to include analysis and inspection methods.

Part I Paragraph 4.1.3 Preliminary Qualification Tests--It is recommended that this paragraph be modified to include analysis and inspection methods.

Part I Paragraph 4.2 Test Requirements--It is recommended that this paragraph be modified to include specification of requirements for analysis and inspection methods other than testing which are included in response to Paragraphs 4.1.1 through 4.1.3.

Part II Paragraph 3.2.1.2 CPC No. 1 Flow Chart--It is recommended that this paragraph be modified to permit the requirement for graphic portrayal to be satisfied by means other than flow charts and to encourage the use of material prepared during program design rather than (possibly) after the fact solely to satisfy documentation requirements.

Part II Section 4 Quality Assurance--It is recommended that this section be modified to be compatible with the broader range of verification methods recommended for Section 4 of Part I.

Part II Paragraph 4.1 Test Plan/Procedure Cross-Reference Index--It is recommended that this paragraph be modified to be compatible with the broader range of verification methods recommended for Section 4 of Part I.

MIL-STD-490 Recommendations

The requirements imposed by MIL-STD-490 are less specific, hence less constraining, in most areas than are the corresponding requirements imposed by MIL-STD-483. Consequently, only minimal revisions are recommended.

B5 Paragraph 4.1 Introduction--It is recommended that this paragraph be revised to establish the requirement for a software quality assurance plan in lieu of the more restricted test plan and test procedures currently specified.

B5 Paragraph 4.2 Test Requirements--It is recommended that this paragraph be revised to include analysis and inspection requirements in addition to test requirements.

DI-S-30567A (CPDP) Recommendations

This Data Item Description (DID) is very comprehensive. In particular, item i, Design Assurance Techniques, is so worded that it easily encompasses any present or foreseeable CPCI or V&V methodology. Thus no recommendations are made for this DID.

SECTION VI

CONCLUSIONS

The new methods of software development, software verification, and system validation have been reviewed to see how they may benefit the development of flight control systems. Four verification methodologies at successive levels of automation were used to illustrate the problems and procedures of verification for a typical triply-redundant flight control system. The specific conclusions drawn from this study are as follows:

1. Software for flight controls is not intricate.

   The observation that the functions of flight control systems are elementary is very important to the discussion. The assembly program for the generic triply-redundant system is about 10,000 lines, but the code may be structured into a large number of simple functions. The data structures are elementary. There are fixed sets of inputs, outputs, and state variables. The control structure is direct with no complicated while-do loops. The control laws require only straight-line computations which have no peculiar singularities that need delicate numerical analysis. The logic functions may be structured as finite-state machines to allow precise design and verification. There is little probability that a construction that causes havoc on very special circumstances will survive a careful inspection on the code. This includes obscure assembly programming manipulations which are easily detected by an independent inspection.

Testing is chiefly used to show that there are no typographical errors in the code and is generally very effective.

The software for high performance aircraft may be more complicated in the future. Examples of new control techniques that require complicated calculations are:

- Adaptive gain mechanizations
- Flutter mode suppression
- Reconfigurable control for battle damage
- Analytical redundancy for sensor failures
- Direct lift control

The verification of these functions will be considered as they are utilized in the flight control designs.

2. Each function may be verified by one or more techniques.

For most of the functions, input and output predicates may be written and the computation may be verified by symbolically evaluating all of the path conditions. Boolean symbolic evaluation of the logic functions is very certain because of the cross-checks afforded by multiple paths. The control laws may be verified with confidence by frequency-response analysis on a simulation. Petri nets may be drawn to check the synchronization of the redundant computers.

3. The integration of functions into the total program may be verified completely.

   The data flows, initialization and integrity of state variables, consistency and lack of circularity of function references, and all of the technical details required in combining the individual functions may be demonstrated by hand or with the assistance of an assortment of tools.

4. The approach to design and verification can be made systematic and formal to any desired level.

   The word formal implies that precise definitions, rigorous analysis, and systematic procedures are written down. Tools may be added until a completely machine-supported approach to verification is achieved. Thus, the technical verification that the software conforms to its design specifications may be done with complete certainty. However, the reliance that must be placed upon the programmer or verifier is directly related to the lack of automation of the methodology. Automation enforces discipline on the design and verification process.

5. Most tools or techniques prove only a particular aspect of the design or code.

   Some are clearly directed at particular properties. The set/use checker and the circular reference checker are examples. Even the more general tools and techniques require explicit and

sometimes implicit assumptions to hold during their use. An automatic theorem prover assumes that the input and output assertions are complete and represent the functional requirements of the segment. Thus, combinations of tools are required and judgment in their use must be exercised. There is always the fundamental assumption that the requirements are complete and have been captured correctly by the top-level mathematical model of the system.

The tools of static analysis can be very useful in the development. Not only may they be used at the coding level but, at the cost of introducing formal languages with syntax, they may be used at the design and specification levels. Since these tools must be constructed to analyze programs in a specific language, their development for general use waits on the adoption of a standard language. None of the usual arguments against using high-order languages for flight control appear to be valid for the new DoD language.

The dynamic tools greatly facilitate and systematize the testing procedures. It has been customary to leave the testing at the function level to the control analyst/programmer. Instrumentation and drivers to monitor and execute tests are generally customized by the flight control analyst to check out his particular program. Again, problems of language and general applicability have slowed the adaptation of testing tools for flight controls.

6. A fully automated design and verification methodology will be constructed eventually.

   This approach will eliminate the need for much self-imposed judgment, care, and discipline in the design and verification process and will certainly reduce the time for development. However, flight control software may be convincingly verified with less completely automated procedures.

7. Validation of the normal system operation is possible.

   The work becomes more difficult as it moves out of the realm of verification of software into the total system. Performance measures may be checked. The computation of the control laws may be verified and validated by checking the frequency response of the system. The normal operation of a flight control system is not so complicated that it cannot be thoroughly checked in simulation. Inductive arguments in this regard are seldom explicitly considered or stated.

8. Validation of the system response to hardware failures is very difficult.

   No techniques were found to help in demonstrating the consistency of the failure management provisions and organizing the analysis of the global performance of the system. It may be possible to validate the effects of a restricted set of failures, but even here no convincing demonstrations were found. It is, of course, not possible to be certain that all events are foreseen.

9. Verification procedures for distributed systems will not be fundamentally different from those for parallel-redundant systems if the assignment of computing tasks are fixed. If reconfiguration is allowed, the problem becomes more difficult.

   These are only speculations because specific architectures for distributed flight control systems are not available. If the software consists of subsystems which are only loosely coupled by executive control, the normal mode of operation between these subsystems will be asynchronous. The system control software to determine the failure status will be more difficult to design and describe. Because of the message-handling protocols, verification of a program may require the simultaneous verification of software and hardware functionality. Validation will require great care.

10. The procurement standards for flight control systems should be revised to include general provisions for software verification and the newer views on software engineering.

These conclusions lead to recommendations for research which are outlined in Section 7.

SECTION VII

FURTHER RESEARCH

The critical need is for more complete and certain methods of validation. While there are needs to work out details and systematic guides for the verification of the technical correctness of flight control software, the theory and practice is understood. Verification and validation procedures are facilitated by structured software design. Research topics in this area are also considered.

VERIFICATION

The methodologies outlined in Section 4 are only intended to provide a structure for considering the directions which the new tools and techniques offer. It would not be difficult to formalize the non-machine methodology and with it to verify a particular flight control system in all details. This will provide a firm foundation to which automated procedures may be added. The analysis may then be extended to fixed and reconfiguring distributed systems.

STRUCTURED DESIGN

It was found that the description of the failure management provisions in the triply-redundant system was not adequate to allow an independent review. The structure hid design decisions and actions of the software to such an extent that verification was not possible. However, by defining the functions as finite-state machines, a sequence of successive refinements could be

made so that all design decisions and assumptions about the mathematical representations of the anticipated events were openly visible. It would be very useful if the limits to the extent of this approach were determined. If the approach could be extended to the total failure management system rather than each of its functions separately, and if it could be automated so that the huge number of anticipated failure modes could be analyzed, the result would be very useful for verifying these complicated processes.

In attempting to apply the SRI-International methodology to the JA-37 system, no essential hierarchy of abstract functionality was found because of the uncomplicated structure of that system and the lack of structure in the data. However, for a triply-redundant flight control system, an abstract functional hierarchy may be possible. If such a structure can be found, it will yield a more efficient and comprehensible design. The verification and validation will then be an easier task. It will be useful to know the extent to which hierarchies of abstract machines may be used in describing segments of flight control systems.

VALIDATION

The process of validation can only be successful to the degree of completeness and accuracy of the requirements. Progress in guidelines for writing requirements has been made by K. Heninger and colleagues[56,57] for avionic

---

[56] K. L. Heninger, "Specifying Software Requirements for Complex Systems," Specification of Reliable Software, Cambridge, Massachusetts, pp. 1-14, April 1979.

[57] K. L. Heninger, J. W. Kallander, J. E. Stone, D. L. Parnas, "Software Requirements for the A-7E Aircraft, NRL Memorandum Report 3876, Naval Research Laboratory, Washington, D. C., November 1978.

software. This work should be applied to flight control systems. The next step is to formalize the statements of requirements so that the validation process may be discussed in a more rigorous manner.

Extended data flow analysis using the power of graph theory and supported by automatic data processing should be useful in modeling the global properties of failure management system. It should be possible to automate the failure effects analysis and to classify failures into equivalent classes.

There are at least two approaches to designing failure management facilities. In the engineering approach, a detailed study of the possible failures is made. From these, self-tests and cross-channel monitoring tests are chosen to give failure coverage and a proper level of reliability for each stage of system degradation. This ad hoc approach may leave some conflicting situations. It is difficult to demonstrate that there are none. In the other direction, only cross-channel comparisons are used to detect failures. This permits rigorous definitions and theorems about how many computers and voters are required for a given number of failures. From an engineering view, these systems are very conservative. Much hardware and software are required. The question remains whether there is a middle ground which takes advantage of the failure mode analysis but can also utilize an extension of the theoretical work.

If practical analytic techniques for demonstrating system validity over a given set of events can be developed, the complementary need is to have a systematic method for constructing validation tests to cover the possibilities omitted by the analysis. A convincing validation can then be obtained with limited testing.

APPENDIX A

SYNCHRONIZATION

Some details of the triply-redundant system are provided here to illustrate the configuration for synchronization, its description in HIPO charts and by Petri net diagrams, and a verification of the halt-release equations by enumeration of states.

THE MECHANIZATION

The configuration for synchronization is shown in Figure A-1. Each block of hardware communicates with the other two blocks and its respective computer. In addition there is a flip-flop which is set by the local computer program when its execution leaves the initialization phase. This flip-flop may be read by the other computers. The corresponding Boolean variables are called the right_, local_, and left_up_and_readys to distinguish them from the ready signals available from the hardware logical shown in Figure A-2.

After the hardware is reset, the real-time counter counts for 25 msec and sets the ready flip-flop. If the counting continues through the overcount period, the overcount flip-flop is set. The hardware produces the halt release signal from the following two terms, which are combined at the final or-gate. For two or three computers,

```
halt_release = ((left_overcount AND local_overcount) OR
right_ready) AND ((right_overcount AND local_overcount) OR
left_ready) AND local_ready
```

but to provide for the case in which two computers fail we need the term

```
halt-release = ((NOT left_overcount OR
NOT local_overcount) AND NOT right_ready)
AND ((NOT right_overcount OR
NOT local_overcount) AND NOT left_ready)
AND local_overcount
```

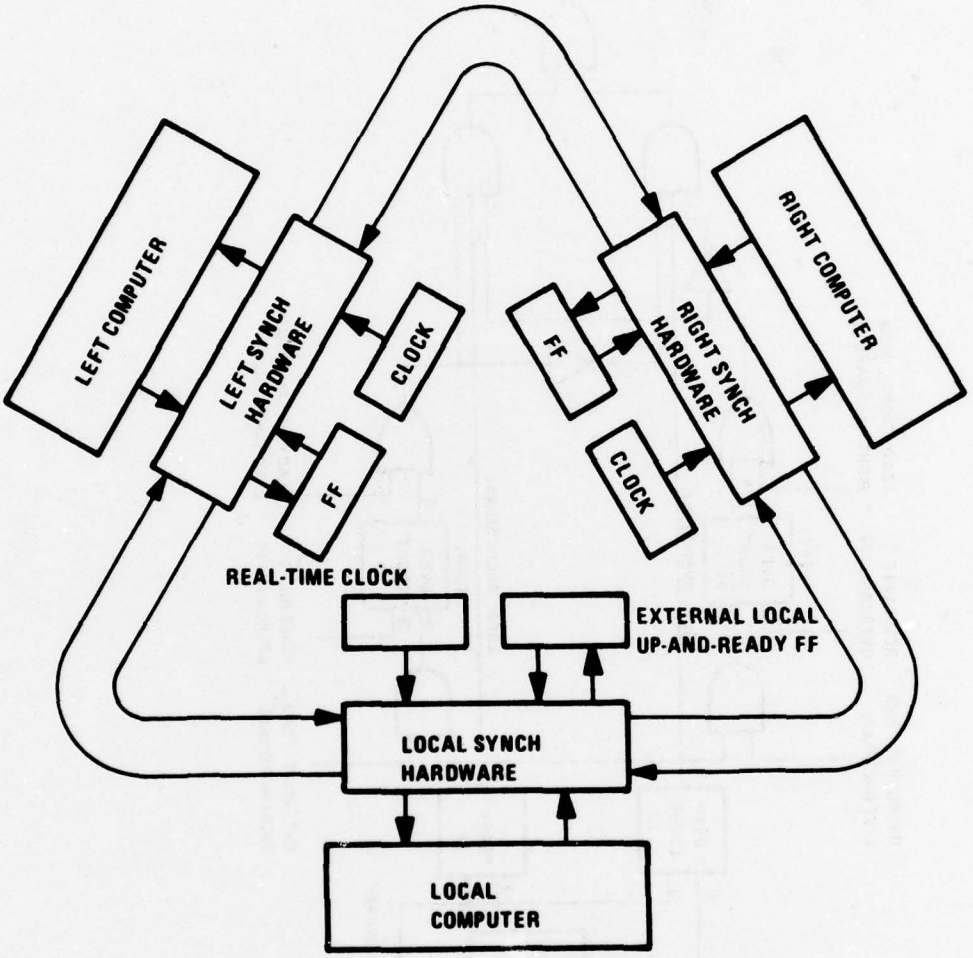


Figure A-1. Configuration for Synchronization

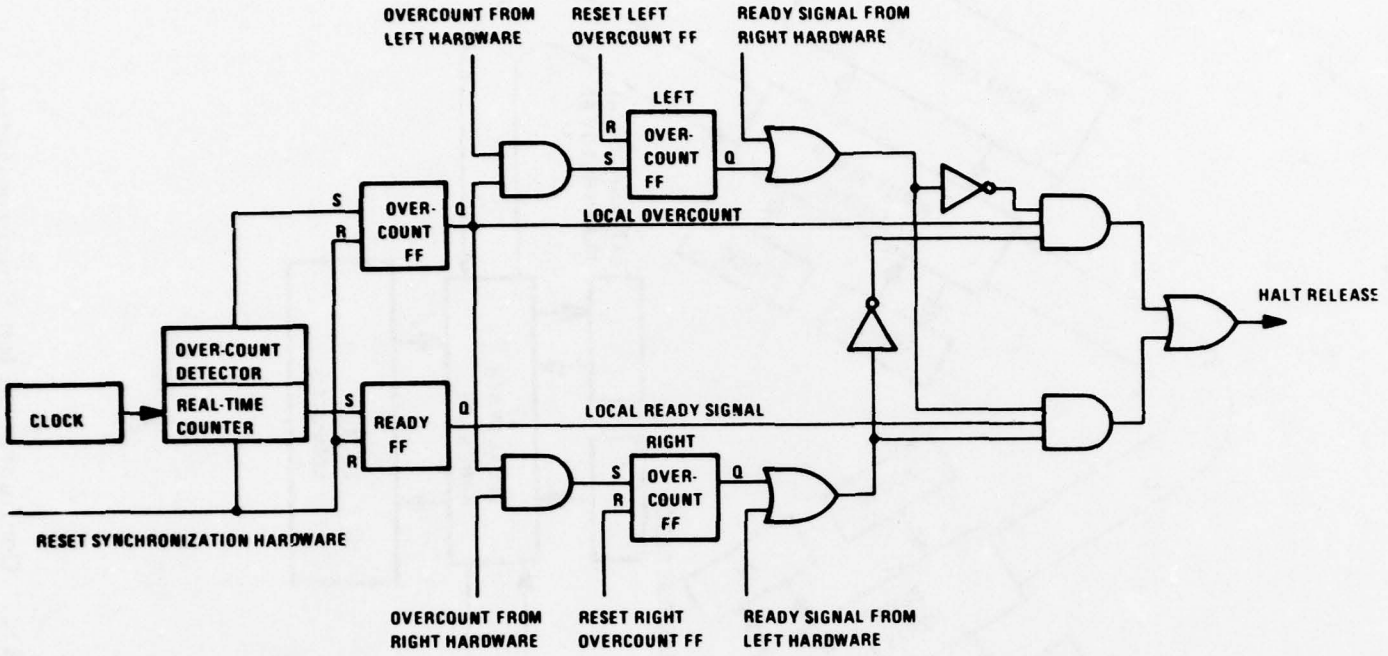


Figure A-2. Hardware Logic for Synchronization

The hardware is implemented so that the power-down or brokenwire case appears as ready = TRUE and overcount = TRUE to the other channels.

The process of initially synchronizing with a computer which is already running in the frame is illustrated in Figure A-3. If the right or left computers are already running, they have passed the instruction in the initial leg of the program that sets the up_and_ready flip-flop. The local computer detects that this flip-flop is set and waits in the starting leg until the running computer passes the halt and resets the ready flip-flop in the synchronizing hardware. This is detected in the wait-loop and the local computer is released. The times required for the program to run from the ready reset to the beginning of the loop and from the wait to the beginning of the loop are balanced to maintain the synchronization.

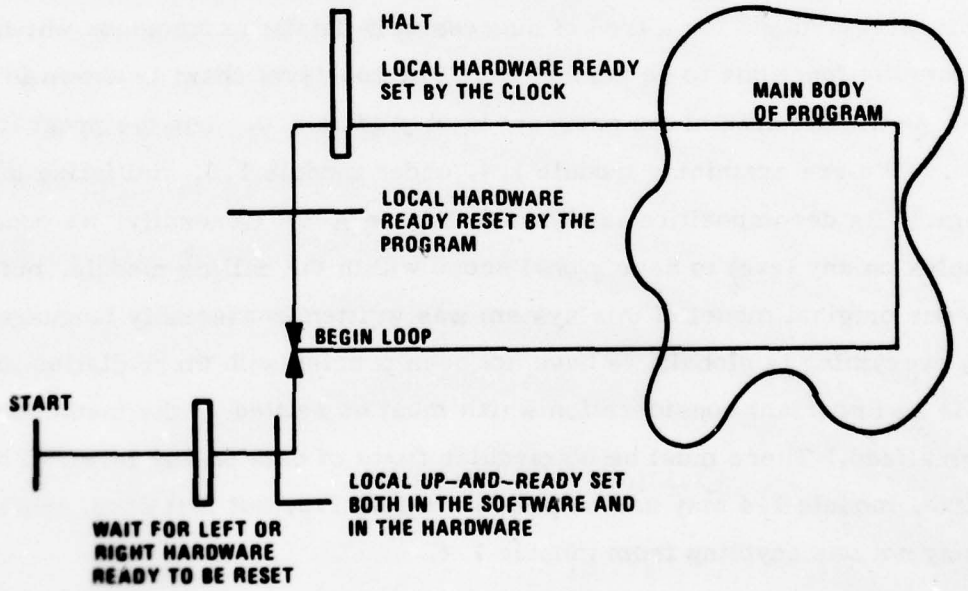


Figure A-3. Initial Synchronization

DESIGN OF THE CODE

To illustrate the use of the hierarchy-plus-input-process-output (HIPO) charts, the design for the module to "initially synchronize the computers" is considered.

The HIPO charts are usually hand-written. One expects changes to be made before and after the design review and typing opens a source of errors. The description should be self-contained and complete. We have added a list of output assertions. This list, when it can be compiled in a reasonable form, is useful in summarizing the actions of the module and in verifying that the code complies with the specifications as represented by those assertions.

The hierarchy stands for a tree of successively detailed commands which elucidate the functions to be performed. The top-level chart is shown in Figure A-4. The bulk of the program is in module 2.0, "run the program frame." We are examining module 1.4, under module 1.0, "initialize the system." Its decomposition is shown in Figure A-5. Generally, we consider variables on any level to have global scope within the calling module, but since our original model of this system was written in assembly language for which everything is global, we have not been precise with these distinctions. This is an important consideration which must be settled as the methodology is formalized. There must be no circular flows of data on any level. For example, module 1.4 may use outputs of module 1.2, but if it does, module 1.2 may not use anything from module 1.4.

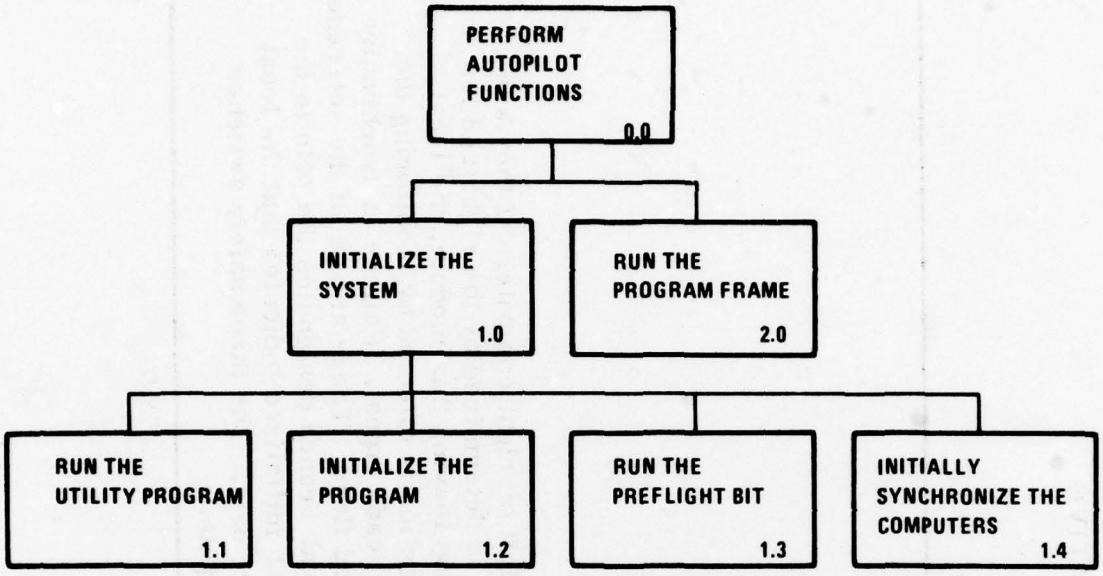


Figure A-4. Top-Level Chart

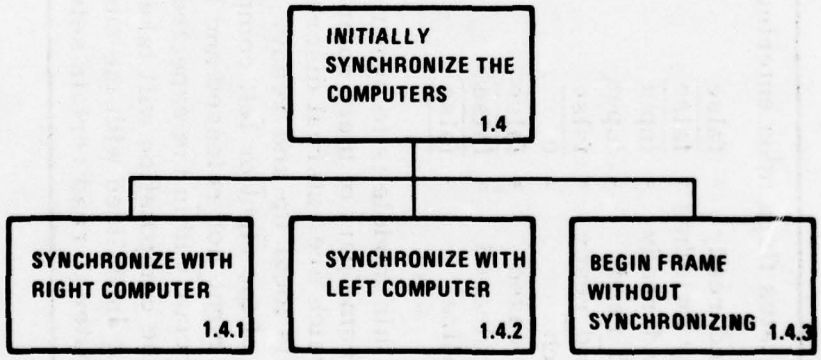


Figure A-5. Initial Synchronization

CHART A-1. NARRATIVE

1.4 initially synchronize the computers (narrative)

assertions on status flags upon entering:

local_up_and_ready	=	false
three safety switches	=	false
right_up_and_ready	=	input
left_up_and_ready	=	input
local_up_and_ready	=	false
local_branch	=	0
left_not_released	=	false
right_not_released	=	false
local_initialize	=	false

The purpose of this module is to synchronize with the left or right computers if they are running in the frame part of their computations. This is determined by the right_and left_ up_and_ready which are external discrete signals. Upon leaving this module, the local computer sets its local_up_and_ready. The synchronization is effected by monitoring the set-reset cycle of the right or left computer hardware ready signal. Failure to synchronize is shown by the right_not_released and left_not_released flags. These are set if the set-reset cycle does not occur within the expected time. The local_branch designates the path in the rate structure the computation will take. The flag local_initialized indicates that the local program must be initialized with the current filter state data. The three safety switches enable the computer to read certain sets of discrete inputs.

CHART A-2. DETAILED CHARTS

1.4 initially synchronize the computers

Input	Process	Output
left_up_and_ready right_up_and_ready (external discrete signals)	if right_up_and_ready is set then "synchronize with right computer" endif if right_up_and_ready is not set or the attempt to synchronize with the right computer was not successful then if left_up_and_ready is set then "synchronize with left computer" endif if left_up_and_ready is not set or the attempt to synchronize with the left computer was not successful then "begin frame without synchronization" endif endif set local_up_and_ready set the three safety switches end (initially synchronize the computers)	watchdog timers set local_branch local_initialized real_time_counter set local_overcount set left_overcount set right_overcount set local_up_and_ready set three safety switches set

CHART A-3. DETAILED CHARTS

1.4.1 synchronize with right computer

Input	Process	Output
right_ready (external discrete from synchronizing hardware)	do wait until the right_ready is reset or the maximum time for release has elapsed if the maximum time for release has elapsed then set the right_not_released flag, attempt at synchronizing with right computer is not successful. else (right_ready is reset) reset local real_time_counter (and local_overcount and local_ready)	right_not_released real_time_counter
real_time_counter max_interval (constant)	do wait until right_ready is set or local real_time_counter has timed up to max_interval if the max_interval has elapsed then set the right_not_released flag, attempt at synchronizing with right computer is not successful. else (right_ready is set) do wait until right_ready is reset reset watchdog timers reset real_time_counter and local_overcount and local_ready. endif	local_overcount local_ready (*)

CHART A-3. DETAILED CHARTS (concluded)

1.4.1 (continued)

Input	Process	Output
real_time_counter	do wait until real_time_counter counts to time_point_01	
	read exec_word_2 from the right computer and extract right interval	
time_point_01 (constant)	set branch in local exec_word_2 to branch from right exec_word_2	local_branch
exec_word_2 from right computer	(interval designates which path in the rate-tree structure is current).	
	Set local_initialized	local_initialized
	endif	
	end (synchronize with right computer)	

CHART A-4. DETAILED CHARTS

1.4.2 synchronize with left computer

Similar to 1.4.1

1.4.3 begin frame without synchronization

Input	Process	Output
	reset real_time_counter and local_overcount and local_ready reset watchdog timers reset left_overcount reset right_overcount end (begin frame without synchronization)	real_time_counter local_overcount local_ready watchdog timers left_overcount right_overcount

CHART A-6. OUTPUT ASSERTIONS

assertions on the status flags upon leaving "initially synchronize the computers"

local_up_and_ready set (external flip-flop)
three safety switch set (enable discrete inputs)

Case: synchronized with right computer

right_up_and_ready	=	true
local_initialized	=	true
right_not_released	=	false
local_branch	=	right_branch

Case: synchronized with left computer

left_up_and_ready	=	true
local_initialized	=	true
left_not_released	=	false
local_branch	=	left_branch
right_not_released	=	right_up_and_ready

Case: begin without synchronizing to left or right computers

local_initialized	=	false
local_branch	=	0
right_not_released	=	right_up_and_ready
left_not_released	=	left_up_and_ready

Charts A-1 through A-6 are the detailed input-process-output charts. These tell what the modules do. The definition of the inputs is evident but the definition of the outputs is less certain. We have generally listed any result as an output whether it is used external to the module or not. This must also be settled when the methodology becomes formal. The process may be described by pseudo-code, flow charts, decision tables, etc., as long as the description is complete and unambiguous.

We can make several observations on the design and the HIPO presentation. The do wait until -- at the (*) has not been guarded against failure of the reset of the ready flip-flop. This may be taken as a design flaw. It is not clear what happens if the three computers begin simultaneously, for example after a restart due to a power transient. This must be considered in some sort of global analysis.

PETRI NETS

Verification may be enhanced by duplicating the description in the HIPO charts by flow charts or some other graphical technique. Petri nets were used to describe the initial synchronizing process.

A Petri net is a directed bipartite graph of alternating vertexes called places and transitions. These are represented in sketches as circles and bars, respectively. The state of the net is determined by markers or tokens on the places. The net may thus be classified as an activity-on-vertex network or as a labeled graph. The major use has been in modeling systems of events in which some events may occur concurrently but with

constraints on the concurrence, precedence, or frequency of the events. To see how this works and to determine its utility, we consider the following example.

Synchronization of Two Computers

A Petri net representing the operation of the local computer with the right computer is shown in Figure A-6. The left computer is not turned on. Places P_1 to P_3 with transitions T_1 to T_3 represent the operation of the local clock. Places P_4 and P_5 with transitions T_3 and T_4 are the states of the overcount flip-flop. Places P_6 and P_7 with transitions T_5 and T_6 are the states of the ready flip-flop. Transition T_6 represents the halt-release logic of

halt-release = local-ready AND (right-ready OR local-overcount)

The local computer is at halt in place P_{10} but is running and crosses the reset command in the software at transition T_{11}.

The synchronization of the local computer with the right computer, which is already running in its frame, is shown in Figure A-7. The inputs from the right computer are the right-ready flip-flop, places P_6, P_7, and transitions T_2, T_5 in the center of the diagram, and the right up-and-ready register represented by places P_1 and P_2. The synchronization occurs upon detecting the right-ready flip-flop being reset by the right computer's software command. If the right-ready is set, we wait for reset at P_9; if it is reset, we must wait for a cycle at P_5 and P_{13}. There are lots of housekeeping needs suppressed at places P_{10}, P_{14}, P_{15}, P_{12}, and P_3.

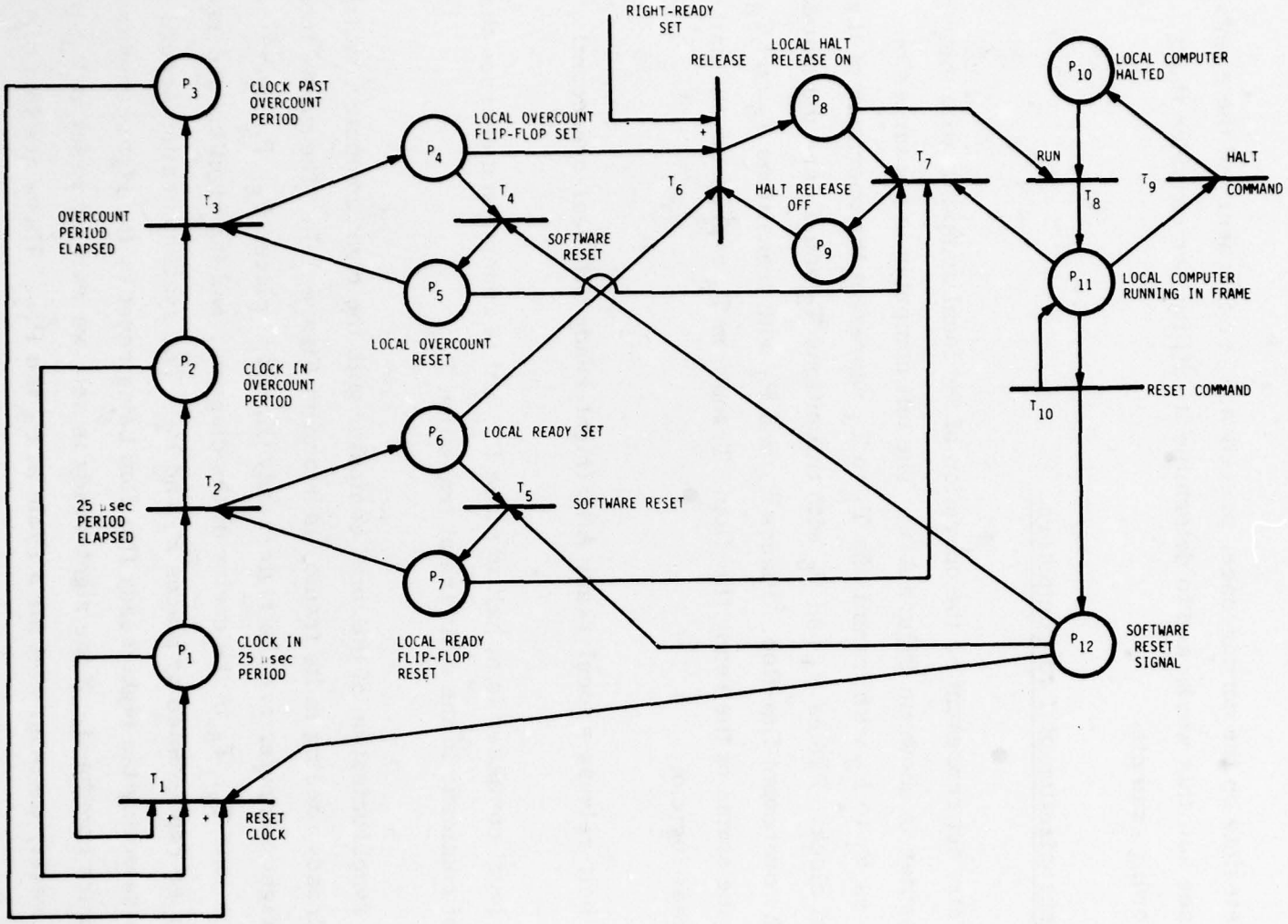


Figure A-6. Petri Net for Local Computer

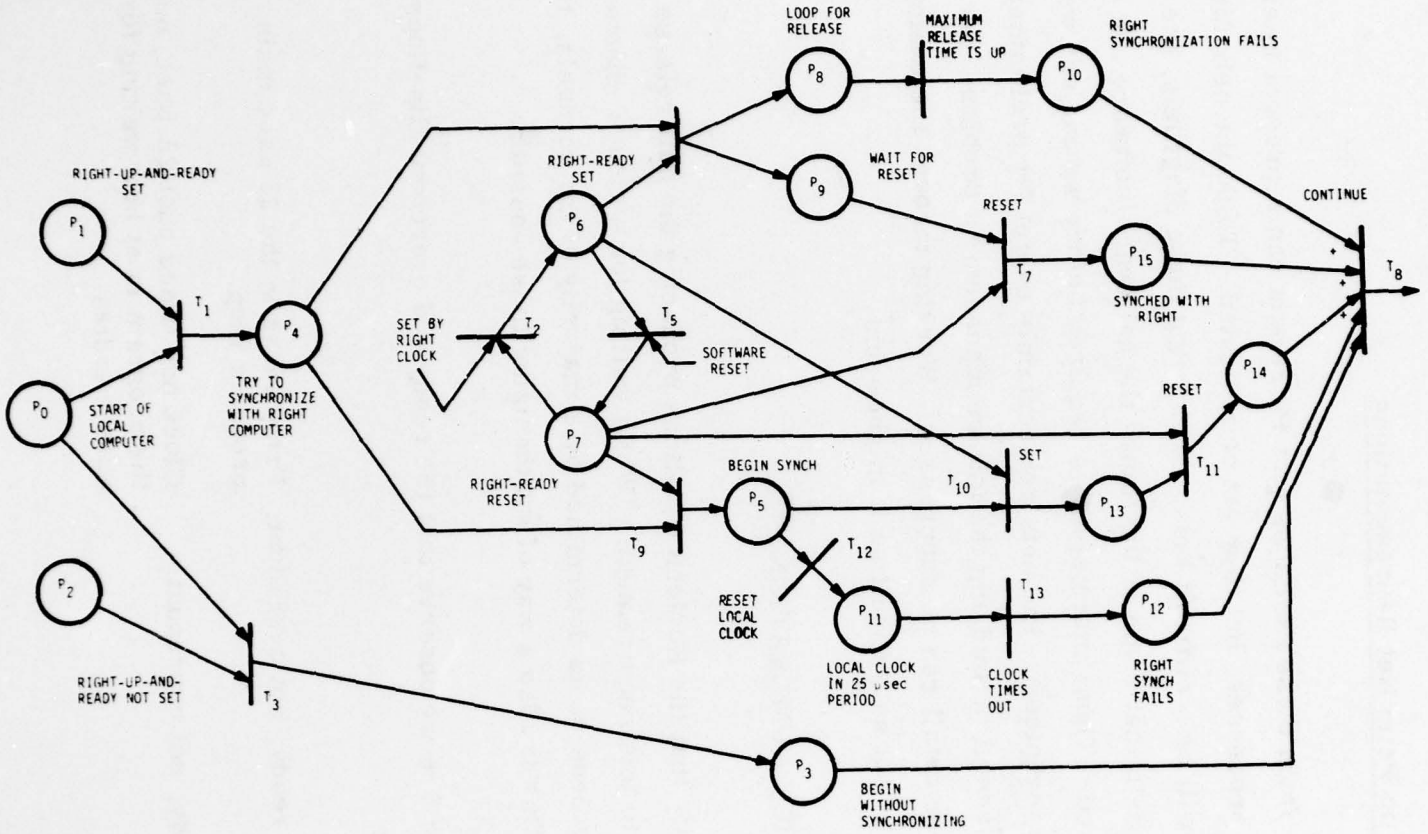


Figure A-7. Petri Net for Initial Synchronization

Comments on Petri Net Representations

We can see from these examples that Petri nets can represent interacting concurrent processes, but they get complicated. To do the nets for three computers will be a difficult task. In drawing these diagrams, the uneasy feeling persists that some of the things the hardware/software can do might be overlooked. Thus, more study is required before we can say we have a *verification technique*. The nets are certainly useful for preliminary design and may be useful in studying the failure management problem, perhaps because more detail can be suppressed. Working through the examples with tokens is not as difficult as it might seem.

SYSTEMATIC ENUMERATION

We now verify that the Boolean equations producing the halt release signals correspond to this requirement: "While waiting for a slower channel, if the wait is excessive as determined by a majority of the channels, then release." This is also a way of synthesizing those equations.

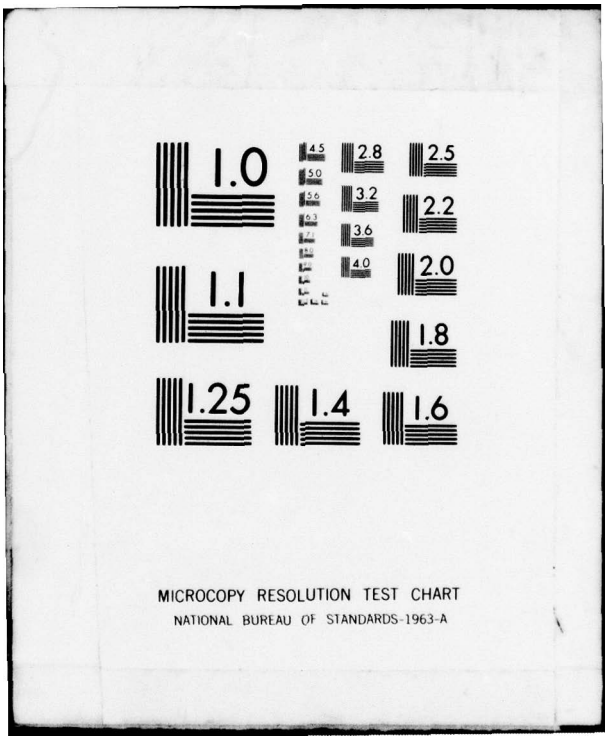
From Figure A-6 we observe that the ready and overcount flip-flops define three states:

1. *not ready, not overcount:* Program is in the 25 μsec main program loop.
2. ready, not overcount: Clock has timed past 25 μsec; normally the program is at halt waiting for the other readies.

AD-A076 021 HONEYWELL SYSTEMS AND RESEARCH CENTER MINNEAPOLIS MN F/G 1/3
DIGITAL FLIGHT CONTROL SOFTWARE VALIDATION STUDY.(U)
JUN 79 E R RANG , M J GUTMANN , D B MULCARE F33615-78-C-3605
UNCLASSIFIED 79SRC18 AFFDL-TR-79-3076 NL

3 OF 3
AD
A076021

END
DATE
FILMED
11-79
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

3. ready, overcount: Clock has counted past the overcount period and has set the overcount flip-flop.

If a computer is not running, then the release logic for the other computers reads that it is in the ready, overcount state. Since there are three computers and three states, there are 27 possible combinations. We can list them all and mark those situations in which a halt release should be issued to the local computer. The combinations are shown in Figure A-8.

The configurations for which the local computer is released are marked I to XI in Figure A-8; this quantifies the requirement. For each configuration the Boolean expression which is true for that configuration may be written as shown in the figure. The disjunction of all of these provides the halt release command, but this may be greatly simplified since each clock overcount that is true implies that the corresponding ready is true Thus,

$$\text{II, VI, VII, IX, } \Rightarrow \text{ I}$$

and we may drop the above four so these two may be omitted

$$\text{X, XI} \Rightarrow \text{VIII}$$

Further, some combine

$$\text{X} \vee \text{IV} = \text{LOCOVCNT} \wedge \text{LEFTOVCNT}$$

$$\text{and XI} \vee \text{V} = \text{LOCOVCNT} \wedge \text{RIGHTOVCNT}$$

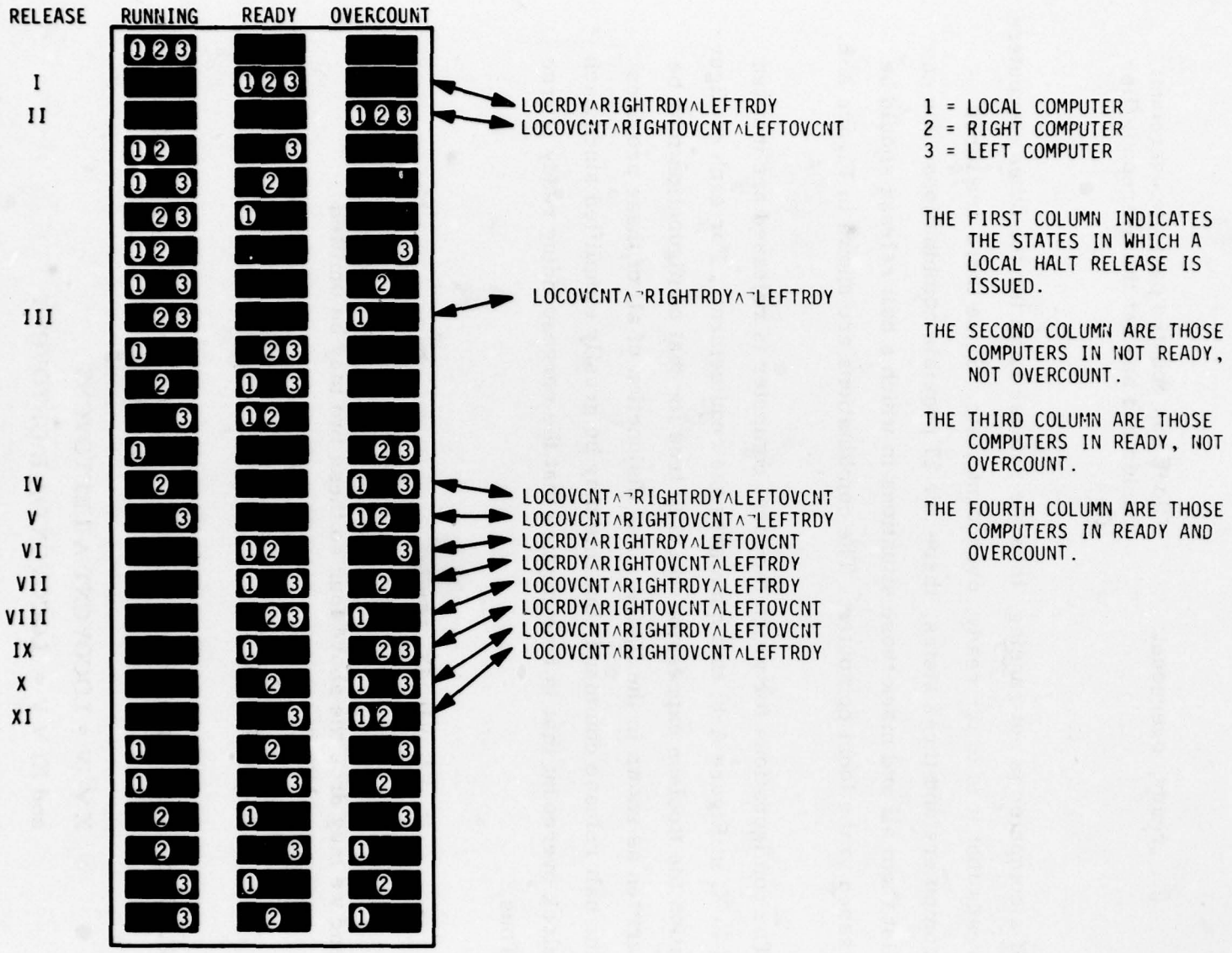


Figure A-8. Enumeration of States

The final result is that

$$
\begin{aligned}
&\text{HALT-RELEASE} \\
&= (\text{LOCOVCNT} \wedge (\text{RIGHTOVCNT} \vee \text{LEFTOVCNT})) \\
&\vee (\text{LOCOVCNT} \wedge (\text{RIGHTRDY} \wedge \text{LFTRDY})) \\
&\vee (\text{LOCOVCNT} \wedge (\neg\, \text{RIGHTRDY} \wedge \neg\ \text{LEFTRDY})) \\
&\vee (\text{LOCRDY} \wedge \text{RIGHTRDY} \wedge \text{LEFTRDY})
\end{aligned}
$$

The final step in the verification is to show that this expression is equivalent to the halt-release logic of Figure A-2. This was done by showing that the two truth tables are equivalent.

APPENDIX B

DESCRIPTIONS OF TOOLS AND TECHNIQUES

The lists and the descriptions of the items are from References B-1 to B-6. Only general descriptions are given. Since each tool or analysis program has its own special set of capabilities, the terminology is not uniform. This complicates compiling lists.

TOOLS

- Accuracy analyzer is a program which analyzes numerical calculations to help determine if they are done with the required accuracy.
- Assembly code verifier is a system of programs which checks for overflow, verifies that assertions about the program are valid, and ensures that there is no self-modification of the instructions.
- Assertion checker proves the theorem that given assertions about the state of a program hold at preassigned points along the computational path.
- Circular reference checker examines the calls between modules to check if one module calls another which in turn calls the first. It also checks for more complicated loops.
- Code comparator checks one tape or card deck against a standard tape or card deck to make certain that there are no differences between the two versions.

- Consistency checker looks for inconsistency between requirements and designs. In the SRI International methodology, the consistency checker examines the relation between two "interfaces," that is, two levels in the hierarchy of development.

- Cross-reference checker checks that for each call to another module the parameter is available as an output, and conversely, if in a module a parameter is declared as an externally referenced quantity, that a module actually calls for it.

- Data base analyzer checks the use of data within a module. It could also be called an internal consistency checker. In some static analyzers, it reports if the module accesses a data base and how it uses or modifies the data base, or if there are unused elements in the data base.

- Documentation and construction systems are large information processing systems to assist and record the development process so that for a large project, programmers have access to a common data pool. Changes and progress may be monitored. In the SRI International system, modified modules must be submitted to the consistency checker before the system will allow a check on the total interface.

- Flow charter shows the logical structure of the program. This may then be compared to the design flow charts to check for discrepancies.

- Formal languages require the format, declarations, and statements of a program to follow definite rules called the syntax of the language. The compliance of the specifications, design, or code to these rules may then be checked for errors.

- Interface checker may check the range and limit of variables to assure that the interface between modules or programs is correct It may also check scaling. In the SRI International system, the word interface denotes the entire program at a particular abstract level and the interface checker performs the cross-reference and circular-reference checks.

- Program flow analyzer provides statistics on the use of statements in the source code and estimates the time required for execution of program elements.

- Set/use checker looks for cases in which a variable is assigned a value but is subsequently not used or the variable is used in an expression but has not been previously initialized or assigned a value.

- Simulations come in many forms. They may be done on a large host computer or on all combinations of hybrid (digital plus analog) or actual hardware systems. They are used to test the characteristics, algorithms, and functions of the project.

- Sneak-path analyzer looks for unexpected paths for information flow through a program by analyzing clues characteristic of sneak paths in the flow graph. We find that in some mode logic designs, erroneous initializations will cause some very peculiar sneaks.

- Standard checker may also be called an editor or possibly a structure analyzer. Its function is to check that the architecture or partitioning rules, documentation conventions, configuration, and data management procedures have been followed. It can check if illegal control structures have been used.

- Symbolic evaluator automatically reconstructs the equations relating the outputs to the inputs of a program. For a program with the logical complexity of a flight control system it is hard to see how the results could be used. However, for individual functions symbolic evaluation would supply a proof of their correctness.

- Test data generator produces test cases to exercise the system. Some produce random inputs while others provide a facility for running standard test cases. Some systems attempt to automatically generate data to exercise all instructions or all branches within a program.

- Test driver controls the execution of a program. A driver may also be part of a program which controls the operation of some external hardware device. A test bed may use actual or simulated hardware to check the overall system operation.

- Test execution monitor collects the data from the counters and interrogation of variables produced by the program instrumentation. It may compare the resulting outputs with those expected.

- Test record generator analyzes and formats the results of the tests. It performs the necessary data reduction or statistical analysis.

- Theorem prover uses stored axioms and theorems to demonstrate that the formal assertions or path conditions generated along the path of the calculation corresponds to the assertion which has been previously specified for that point in the program.

- Timing analyzer monitors and records the time it actually takes to run specified functions and routines of the program.
- Units consistency checker checks expressions to determine that the units declared for the variables produce consistent terms.
- Unreachable code detector looks for code which cannot be executed with any acceptable input data.
- Verification condition generator calculates the symbolic condition that determines if the program will be executing a segment of code and what the relationships between variables will be on that path.

TECHNIQUES

- Abstractions and hierarchies to reduce complexity are advocated in most programming methodologies to make the design simple and clearly defined. There are at least two possible dimensions for hierarchies. Functions may be defined as hierarchical trees with the more detailed computations done in the lower levels. There is also a hierarchy of abstract machines in which each lower level supplies a more detailed computational capability, until at the lowest level the instruction set of the computer describes the capabilities of that level of machine. Abstract functions and abstract data structures allow specification of designs without getting lost in the details. We have not yet been able to set up the flight control software with any useful level of abstraction.

- Checkout testing is the testing of each function or each module of the code before they are all integrated in the complete program. These tests are usually left to the coder. Experience shows that the tests are often not done adequately: too many errors are passed on to the integration phase.

- Constructive design approaches follows Dijkstra's ideas of using a formal design language, using predicate transformations to describe the specifications on how the software relates the outputs to the inputs, and using formal assertions to describe loop invariants and path conditions. This provides a proof of the algorithm as it is designed.

- Critical design review is the oral demonstration of the detailed software design before coding and checkout begin.

- Data flow diagram, structure chart are two diagrams which help lay out the preliminary software structure. The data chart shows the flow and transformations of data through the program. The structure chart shows the functional hierarchy. It gives a picture of the design and organization of the program.

- Descriptions or documentation may take many forms. The traditional assembly listing and flow charts are seldom adequate. A new approach is to include the design language or pseudo-code formulation as an insert on the assembly listing. Formal languages have been used at each stage of development. There are many specialized forms to describe special situations.

- Design guidelines, test guidelines, and coding guidelines aid the programmer in following good practices but do not bind him to any particular procedure. He is free to deviate where needed and to choose the most expeditious approach. Standards impose rules which are to be followed without deviations.

- Design standards, coding standards are rules governing languages, formats, control structures, data structure, organization of design, and so on, set down for the programming group.

- Functional capabilities list heads each module description to define the functions the module must perform. It can be used to help define the checkout tests.

- Integration testing is the testing of the code after all modules have been assembled. All module interfaces must be checked as well as the input/output structures.

- Organization as finite automata helps provide a clear and natural structure to many of the functions for flight controls. This is particularly true of functions with high logical content such as mode and switching logic, signal selection, and failure management. The state must be defined, the input calculations and output calculations formulated, and the state transitions must be described. This formulation may easily be checked by reviews or exhaustive testing.

- Qualification audit, also called the functional configuration audit, is a review of the design and the results of qualification testing to make sure that the software is ready for delivery to the customer. Both hardware and software have been tested, integrated, and then tested as a system.

- Singularities and extremes testing explores the limits of the ranges of input data normally expected for the software and test any critical values which may be specific to the algorithm or which may be ruled out as forbidden values in the algorithm. Experience shows that the extremes or the singularities can cause difficulties in software. Fortunately for flight controls, few such problems are found.

- Symbolic execution is also listed as a tool, but it can be performed by hand on special functions such as mode logic to get a proof of design or code.

- Systems concept review is an oral demonstration of the initial concepts, hardware/software trade-offs, and system organization. No hardware or software designs have been made at this point.

- Validation testing is the final demonstration in a simulated environment that the combined operational hardware and software system meets its operational requirements.

APPENDIX B REFERENCES

B-1. Reifer, D.J., and Trattner, S., "A Glossary of Software Tools and Techniques," Computer, Vol. 10, No. 7, pp. 52-61, July 1977.

B-2. Benenati, C.J., and van Dam, A., "An Informal Survey of Software Engineering Tools and Methodologies," Raytheon Submarine Signal Division, Providence, Rhode Island, December 7, 1977.

B-3. Miller, E.F., "Program Testing Tools--A Survey," presented at MIDCON, Chicago, Illinois, November 8, 1977.

B-4. Fairley, R.E., "Tutorial: Static Analysis and Dynamic Testing of Computer Software," Computer, Vol. 11, No. 4, pp. 14-23, April 1978.

B-5. Huang, J.C., "Program Instrumentation and Software Testing," Computer, Vol. 11, No. 4, pp. 25-32, April 1978.

B-6. Panzl, D.J., "Automatic Software Test Drivers," Computer, Vol. 11, No. 4, pp. 44-50, April 1978.

APPENDIX C

AN EXAMPLE OF VERIFICATION

A small segment of code is verified as an example of the process of checking the design to the requirements and the code to the design. In this case, the input assertion is true and the output assertions are easy to state and to check; the verification proceeds by symbolic evaluation.

Only by working through the low-level details of this example and all the other function components of the flight control system can we be certain that the problem of verification has been adequately studied.

A failure mode effects and criticality analysis is outlined. This quickly leads us into questions of system design philosophy. Indeed, the function we are analyzing checks for a computer failure mode which has devastating effects on the process. If this failure mode occurs, it is unlikely that we would even get to this test in the program.

VERIFICATION OF TEST OUTPUT-FROM-A

The function to be analyzed is "test output-from-A." This is a test of the computer-digital/analog-analog/digital-computer hardware links. A narrative description is given in Chart C-1; it is considered the requirement statement. While the program for this test is very short (only 16 lines of assembly code), there are many points to be made from the study. Of note is the detailed knowledge of hardware functions that is needed.

CHART C-1. NARRATIVE

test output-from-A

Several tests are made. On the first output command the digital/analog converter should report ready while on the second request it should report not ready. The 301A has a failure mode in which the contents of the index register are erroneously added to the OCP address. This is tested. Finally, the wrap-around test in which the contents of the digital-to-analog output are read back into the A-register to be compared to the original output is performed. Failure of the index/OCP address test is detected by the wrap-around test.

The verification methodology suggests a HIPO description of the design with symbolic evaluation to verify compliance with the requirements. The design is written in Chart C-2; the analysis is given in Chart C-3. The double representation by pseudo-code on the HIPO chart and by the flow chart in the analysis gives confidence that the design meets the requirements. In this case the symbolic evaluation is best done in a narrative fashion rather than by attaching symbols to everything.

The assembly code is shown in Chart C-4. Its analysis is diagrammed in Chart C-5. The path conditions are written in the righthand column. We may compare the program flow with the HIPO analysis and compare the corresponding path conditions to be certain that the assembly code implements the design.

The decision table analysis is given in Chart C-6. This provides another systematic approach to dissecting the pseudo-code. From it, tests may be set up to activate all of the decision rules and thus provide good test coverage of program functions. The suggested test scenarios are written in Chart C-7.

The program is small. The study may be criticized as much effort spent on a program which could be adequately verified by a careful oral walk-through. On the other hand, we can project that a more complicated module can be worked in a similar manner. The analysis forces documentation to be clear and precise. Since this self-test is used for many different systems, the effort does not seem unreasonable.

CHART C-2. HIPO

test output-from-A

Input	Process	Output
COMMAND	output COMMAND to D/A converter if digital/analog converter is not ready then exit to CPUFAULT else output COMMAND if digital/analog converter is ready then exit to CPUFAULT else load index register with 03777 call wrap-around test initiation read OUTPUT if absolute value (OUTPUT -COMMAND) ≥ ota_tolerance) then exit to CPUFAULT endif endif endif end test output-from-A	COMMAND
ota_tolerance (constant)		
OUTPUT		

CHART C-3. ANALYSIS

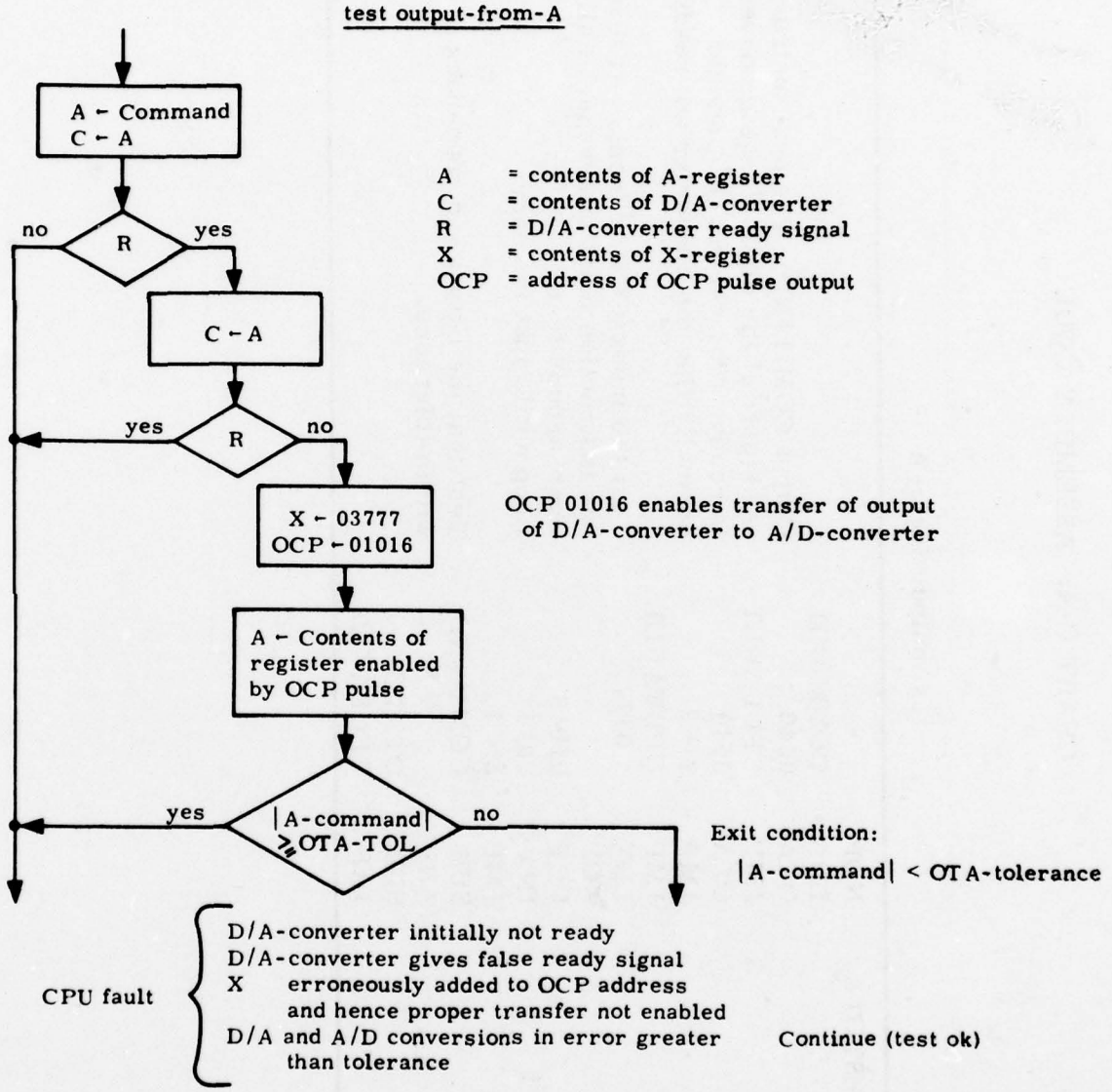


CHART C-4. ASSEMBLY CODE

test output-from-A

```
TESTOTA   NOP
          LDA     COMMAND
          OTA     0040          Output COMMAND.  The next instruction
          JMP     CPUFAULT      is skipped if the digital/analog converter
          OTA     0040          is ready.  The second OTA should
          JMP     $ + 2         normally find the converter not ready.
          JMP     CPUFAULT
          LDX     = 03777       Test the index/OCP address.  At least
          NOP                   one instruction must intervene.  Call
          OCP     01016         wrap-around test.
          INA     0014          Loop until input is ready.
          JMP     $ - 1
          SUB     COMMAND       Test that the output and read-back is
          ABS                   within tolerance.
          SUB     OTATOL
          JAP     CPUFAULT
```

CHART C-5. ANALYSIS OF ASSEMBLY CODE

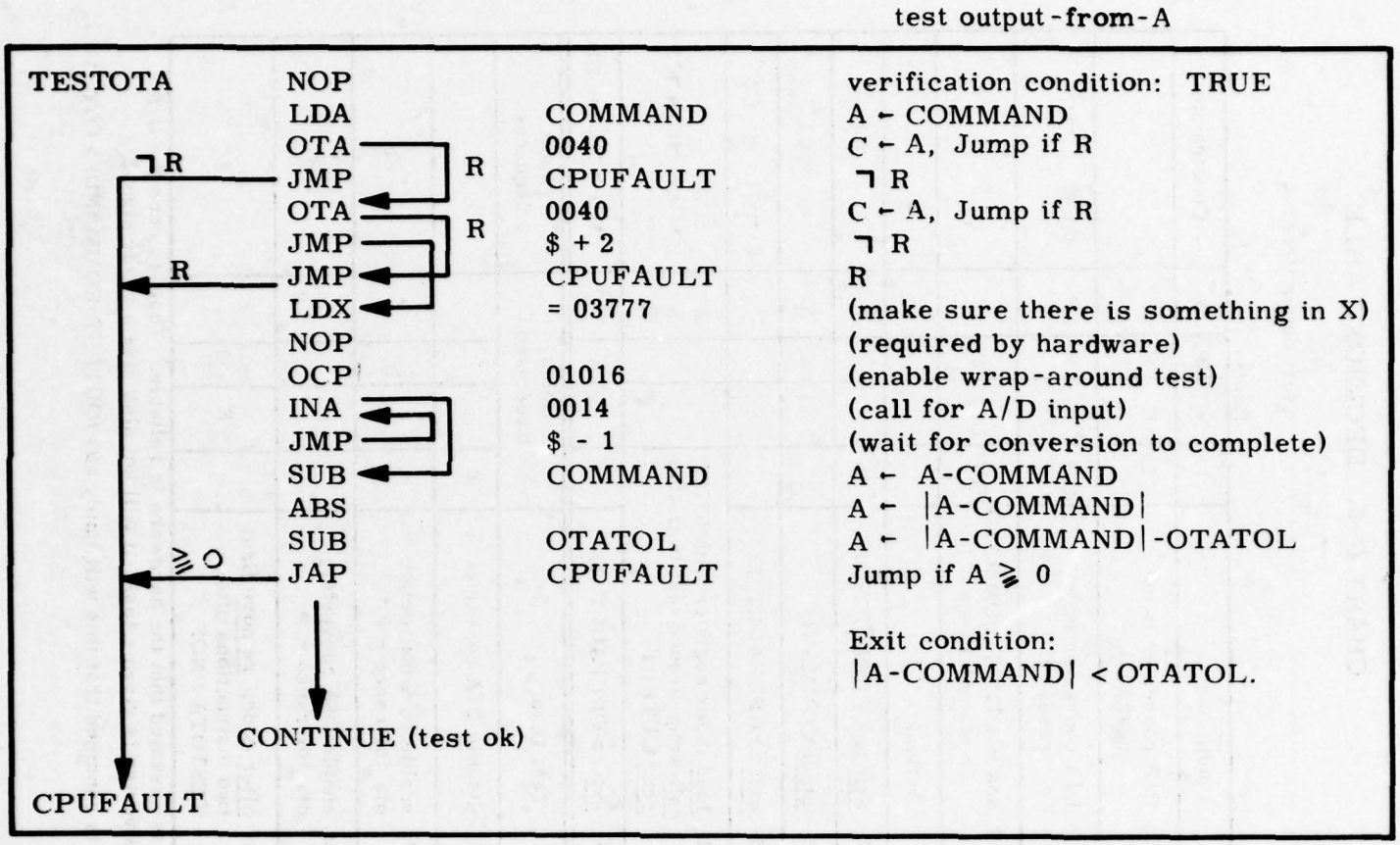


CHART C-6. DECISION TABLE

test output-from-A

Conditions	Rules 1	2	3	4	Constraints
C1: D/A converter is not ready (1)	T	F	F	F	
C:2 D/A converter is ready (2)	D/C	T	F	F	C1 $\not\equiv$ ¬C2
C:3 abs (OUTPUT-COMMAND) > ota_tolerance	D/C	D/C	T	F	
Actions					
A1: exit to CPUFAULT (1)	X				C1
A2: output COMMAND		X	X	X	¬C1
A3: exit to CPUFAULT (2)		X			¬C1 ∧ C2
A4: load index register with 03777 call wrap-around test initiation read OUTPUT				X X X	¬C1 ∧ ¬C2 ∧ ¬C3
A5: exit to CPUFAULT (3)			X		¬C1 ∧ ¬C2 ∧ C3

Test Data	Rules Exercised				Rule Sequence
D1: disable D/A converter	X				1
D2: enable D/A converter; * ota_tolerance ← 0 †			X		3
D3: enable D/A converter; ota_tolerance ← ∞				X	4
D4: alter code: remove first two instructions after TESTOTA NOP		X			2

*It is assumed that the hardware is reliable. That is, once the D/A converter is given a task, it will not also give a ready signal.

†It is assumed that this will imply abs (OUTPUT-COMMAND) > OTATOL.

CHART C-7. TEST SCENARIOS

test output-from-A

Test I:	disable D/A converter execute module investigate: has a correct jump to CPUFAULT been made?
Test II:	enable D/A converter ota_tolerance ← 0 execute module investigate: has a correct jump to CPUFAULT been made?
Test III:	enable D/A converter; ota_tolerance ← ∞ execute module investigate: has the program CONTINUED correctly?
Test IV:	alter code: LDA command ← NOP OTA 0040 ← NOP enable D/A converter execute module investigate: has a correct jump to CPUFAULT been made?

Some items in the assembly code cannot be checked by this approach; they are dependent on the architecture of the machine. The NOP between the LDX and the OCP instructions is required for the test. We, the verifiers, do not know the reason. Its omission would go undetected. The loop back after the INA instruction would be flagged by an inspector as a program error or at least as bad practice. The A/D-conversion must be allowed to complete. Note that a hardware fault could cause the machine to hang up at this point. The mechanism of the ready signal on the OTA instruction is a hardware detail checked by the program but its exact operation requires more explanation. We conclude that complete verification of this test must further examine the hardware level.

CRITICALITY ANALYSIS

The function does not compute variables for subsequent use by other modules; rather, its exit conditions are its outputs. A normal exit reflects satisfactory test outcome, whereas an exit to CPUFAULT reflects an unsatisfactory test outcome. The only other possibility is failure to exit, caused by a nonterminating loop at

```
INA 0014
JMP $-1
```

Thus the failure modes of this module are:

- Exit to CONTINUE when a hardware fault has occurred
- Exit to CPUFAULT when no fault has occurred

- Failure to exit
- Jump to some random instruction not in the module

The criticality analysis considers the effect of each of these modes. Table C-1 rates the process.

False Exit to CONTINUE

The analysis and test cases of Chart C-6 confirms that a false exit to CONTINUE can only occur if one of the following has also occurred:

1. There is a hardware failure within the computer that causes anomalous control flow.
2. There is a hardware failure in the computer that causes the contents of the A register to be falsely negative at the last instruction in the module.
3. There is a failure in the digital-to-analog or analog-to-digital conversions which is not revealed by the value of COMMAND on this iteration.

The first two of these possibilities are potentially severe. It is much more likely that these types of hardware failures will have wider effect, causing massive computer failure. At this point, an interface with the hardware analysis is needed to assess the potential for such subtle hardware failure modes.

TABLE C-1.

ANALYSIS OF CRITICALITY

Outputs			Software Failure Mode					Assurance Method Protection
Variable	Type	Used By	Description	Type	*Enabling* Conditions	Effect	Severity	
(Note: Exit states constitute output)	DNA							
Exit to continue (normal exit)		Flow enters "Test Intercom"	Exit to "Test Intercom" when failure has occurred	Generic software	None	Loss of fault detection	Extreme; all channels affected	Symbolic evaluation; test cases derived from decision tables.
				Software/ hardware	Hardware control flow failure	Loss of fault detection	Moderate; single channel affected	Refer to hardware analysis for likelihood of failure.
				Software/ hardware	Hardware A Register sign bit stuck at 1	Loss of fault detection	Moderate; single channel affected	Refer to hardware analysis for likelihood of failure.
Exit to CPU Fault		Flow enters CPU Fault	Exit to CPU Fault when no failure has occurred	Generic software	None	Good channels condemned	Extreme; all channels condemned	Analysis: Transfer not subject to subtle variable values; testing will detect if present.
Failure to Exit			Nonterminating loop at INA 0014 JMP$-1	Generic software	None	Software hang-up	Extreme; all channels inoperative	Analysis: Loop termination requires completion of INA with any value. Any test will detect if present.

The third possibility, an undetected failure in the conversions, also requires consideration of the hardware analysis. Usually the failure mode would be revealed by a different value of COMMAND after a short time, so the failure would be detected.

False Exit to CPUFAULT

A false exit to CPUFAULT results in an operable computer being declared failed. If the exit is caused by a software error, the results are particularly severe because all channels of a redundant system would be disabled. It would be a common-mode failure. Such an error would manifest itself repeatedly during testing, so its survival into operational software is unlikely.

Failure to Exit

The failure of the program to exit because the read-input loop does not terminate could result from an inherent software deficiency or from a hardware failure condition. If caused by an inherent software deficiency, this failure mode would be common to all channels. It would consequently be very severe. Since this portion of the code would be exercised during testing and since it is independent of data values, the possibility of this failure mode being present as an inherent software deficiency can be dismissed.

The more interesting possibility is that the mode results from a computer hardware failure. The other channels would correctly judge the failed channel to have failed when they did not receive a ready signal at the next

synchronization attempt. Thus, this program behavior might or might not be considered a fault or failure, but in any event its severity would be the same as any other loss of the computer channel.

Jump to Random Instructions

A jump to some random instruction not in the module can occur as a generic software error or as the result of a hardware error. The possibility of a generic software can be dismissed by the time the verification process has been completed. The failure mode resulting from hardware failure must be addressed by hardware analysis.

Evaluation of Criticality Analysis

As illustrated by the example, the criticality analysis must be based on the structure of the code to a minimal extent. Otherwise, the analysis tends to duplicate verification analysis: generally code review. This in turn tends to violate the fundamental concept of the analysis, which is to exhaustively examine failure modes and their effects and criticality. In some modules this approach may not be feasible in that the statement of failure modes cannot be sufficiently separated from the code implementation.

Another difficulty arising in practical application to software is the difficulty of defining what constitutes adequate separation of the analysis from the structure. The structure must be considered in defining the potential failure modes, but the extent to which code details should be allowed to enter the analysis is not clear.

It may be noted in the foregoing that anomalous behavior of the test as a result of software error is well covered by the verification methods proposed. The greater area of concern must be the behavior of software in the presence of hardware failures, in particular those failures which could lead to erratic computation but which do not totally stop the computer from executing. This is an area which has not been treated fully and is appropriate for research beyond the scope of this report.

APPENDIX D

AN ALGORITHM FOR SELECTING SIGNALS

This example is included to show that if the need for verification is considered during design, clear and clean structures will result.

Our goal is to have a software design for the failure management of a triply-redundant system which may be verified by oral demonstration and for which certain aspects may be verified by more formal means. The primary functions, as illustrated in Figure D-1, are:

- Select signals
- Perform self-tests
- Monitor outputs
- Check consistency

This appendix considers the select signals function. The following claims are made:

1. Specification of the module as a finite state automaton, Table D-1, is precise, complete at the top-level, and easy to work from.
2. The substitution tricks used in previous designs are unnecessary, obscure the design, and make verification difficult.
3. A straightforward design is ultimately the most efficent design.

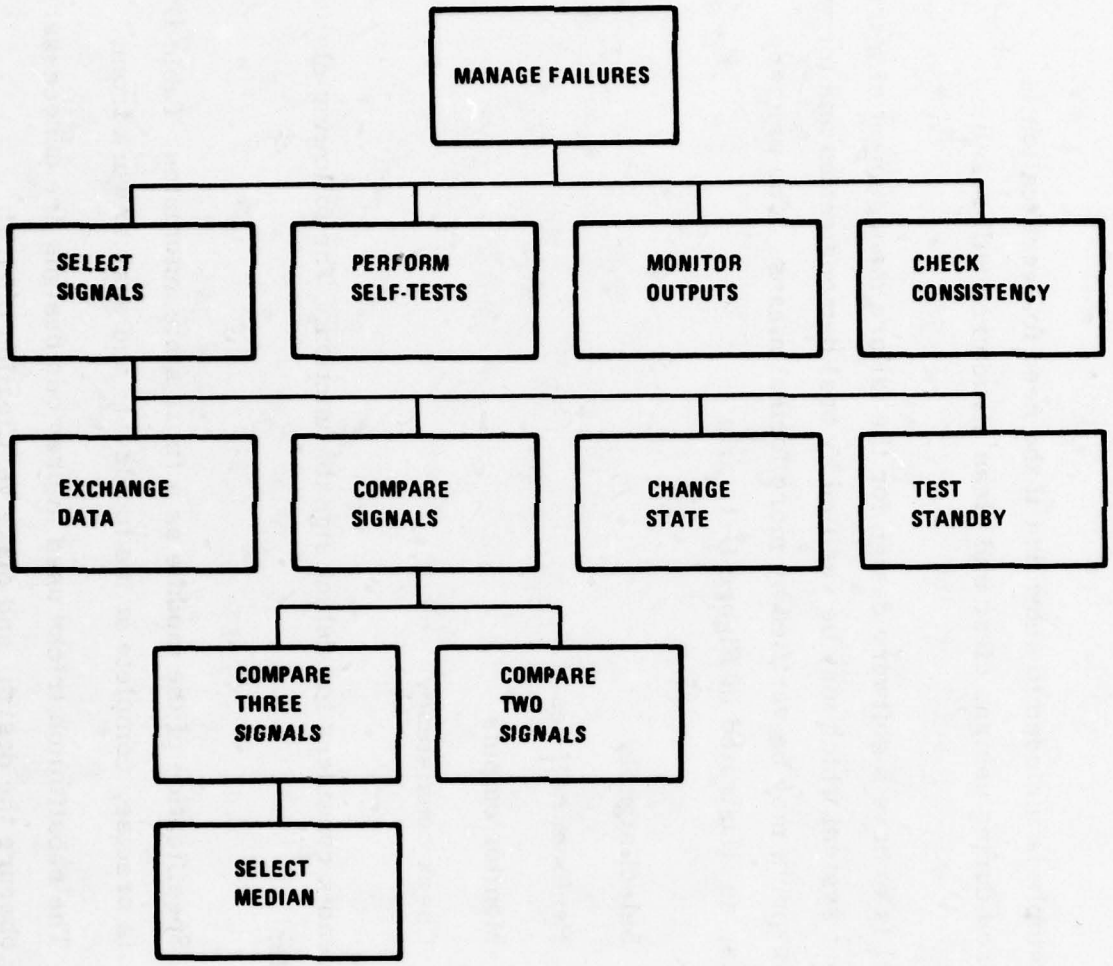


Figure D-1. Hierarchy

TABLE D-1.

SPECIFICATION OF FAILURE STATE AUTOMATION

States \ Events	left computer goes down	left computer comes up	right computer goes down	right computer comes up	left sensor fails	local sensor fails	right sensor fails	standby sensor fails	standby sensor recovers	state of left computer	state of right computer
Three Computers Running											
0 no sensor failures	30	--	20	--	1	2	3	4	--	0	0
1 left failed	30	--	21	--	--	5	7	8	--	2	3
2 local failed	32	--	22	--	5	--	6	9	--	3	1
3 right failed	33	--	20	--	7	6	--	10	--	1	2
4 standby failed	34	--	24	--	8	9	10	--	0	4	4
5 left and local	32	--	27	--	--	--	17	17	--	6	7
6 local and right	37	--	22	--	17	--	--	17	--	7	5
7 right and left	33	--	21	--	--	17	--	17	--	5	6
8 standby and left	34	--	27	--	--	17	17	--	1	9	10
9 standby and local	37	--	27	--	17	--	17	--	2	10	8
10 standby and right	37	--	24	--	17	17	--	--	3	8	9
17 three sensors failed	STOP										
Left and Local Running											
20 left, local and standby ok	40	--	--	0	21	22	--	24	--	30	--
21 left failed	40	--	--	1	--	27	--	27	--	32	--
22 local failed	47	--	--	2	27	--	--	27	--	33	--
24 standby failed	47	--	--	4	27	27	--	--	20	34	--
27 two of left, local and standby	STOP										
Local and Right Running											
30 local, right and standby ok	--	0	40	--	--	32	33	34	--	--	20
32 local failed	--	2	47	--	--	--	37	37	--	--	21
33 right failed	--	3	40	--	--	37	--	37	--	--	22
34 standby failed	--	4	47	--	--	37	37	--	30	--	24
37 two of local, right and standby	STOP										
Local Computer Alone											
40 local and standby ok	--	20	--	30	--	47	--	47	--	--	--
47 local or standby failed	STOP										

4. It is not easy or necessary to add output assertions to every module. For some functions, formal assertions are easy to write. In this case, an output assertion is that all of the transitions meet the specifications of Table D-1. This tells us nothing new.

5. It is useful to reflect on what structure may be imposed on the data of the module. This at least forces one to organize the variables and constants and to think of definitions.

6. The HIPO approach, while somewhat redundant, gives an easily tabulated report and check on the data flow.

7. A pseudo-code similar to Pascal or Algol is useful. It has been suggested that the procedure of subsequently coding the design in Pascal and testing it at that level before converting it to assembly language will be useful.

8. A Petri net is not a useful manual representation of this automaton. But the automaton can be easily represented as a bipartite graph and this representation could then be used as a machine model against which the final code could be automatically verified.

The Design

Most of the transitions of Table D-1 are trivial tautologies. Only a few design decisions are made at this level. They can be stated explicitly.

- There are nine events which affect the state. The standby sensor is the only sensor which is allowed to recover. It is evaluated as long as there are at least two sensors with which it can be compared.

- The dedicated left and right sensors are assumed operative as their respective computers come up.
- Two sensor signals are required for operation. This is reflected in the table of signal selections in Table D-2 and in the automaton.
- The order of sensor failures is immaterial. This is a change from the A-V/STOL design.

The state of the computer/sensor combination is recorded by the flag, local_computer_state. The details of its implementation may be deferred until coding.

If the local sensor has failed, erroneous data will be sent to the other computers but it will never be used. This differs from the generic design, in which the standby is substituted. Our substitutions are made as the functions for processing two or three valid signals are called. The count-down mechanism has not been changed.

The event that a sensor has failed is defined by its counter reaching zero. The addition or loss of the left and right computers is detected from the respective discrete ready signals.

The design is shown in Chart D-1.

TABLE D-2.

SIGNAL SELECTIONS

State		Selection
Three Computers Running		
0	no sensor failures	median (left, local, right)
1	left failed	median (standby, local, right)
2	local failed	median (left, standby, right)
3	right failed	median (left, local, standby)
4	standby failed	median (left, local, right)
5	left and local failed	average (standby, right)
6	local and right	average (standby, left)
7	right and left	average (standby, local)
8	standby and left	average (local, right)
9	standby and local	average (right, left)
10	standby and right	average (left, local)
Left and Local Running		
20	left, local, standby ok	median (left, local, standby)
21	left failed	average (standby, local)
22	local failed	average (standby, left)
24	standby failed	average (left, local)
Local and Right Running		
30	local, right, standby ok	median (local, right, standby)
32	local failed	average (standby, right)
33	right failed	average (standby, local)
34	standby failed	average (local, right)
Local Running		
40	standby and local ok	average (standby, local)

CHART D-1. DETAILED CHARTS

select signals (data structure)

For each set of sensors

values: left, local, right, standby

failure counts: left_count, local_count, right_count, standby_count

status flags: left_ok, local_ok, right_ok, standby_ok

constants: tolerance, max_count, standby_factor

output quantity: selected_signal

module state: local_computer_state

- left_computer_running
- right_computer_running
- left_ok
- local_ok
- right_ok
- standby_ok

the failure counts and selected_signal are also state variables

CHART D-1. DETAILED CHARTS (continued)

select signals

Input	Process	Output

for each
sensor
 	
local_computer_
state
 	
left
local
right
standby
standby_count
left_count
local_count
right_count
 	
selected_signal
 	
left_ready
right_ready
(external discrete
signals)
	for each sensor do
	"exchange data"
	"compare signals"
	"change state"
	if local_computer_state is 0, 4, 8, 9, 10, 24, 34
	then
	"test standby"
	endif
	end for
	end (select signals)
		for each
		sensor
		
		local_computer_
		state
		
		standby_count
		left_count
		local_count
		right_count
		
		selected_signal

CHART D-1. DETAILED CHARTS (continued)

exchange data

Input	Process	Output
selected_signal	past_value ← selected_signal (hold previous output)	past_value
local_sensor	read local_sensor and store as local	local
standby_sensor	read standby_sensor and store as standby	standby
	send local to left and right computers	
local_computer_state	if local_computer_state is 0 to 10 (three computers running)	
	then read local from left computer and store as left	left
	read local from right computer and store as right	
	else if local_computer_state is 20 to 24	
	then read local from left computer and store as left	right
	else if local_computer_state is 30 to 34	
	then read local from right computer and store as right	
	endif	
	endif	
	endif	
	end (exchange data)	

CHART D-1. DETAILED CHARTS (continued)

compare signals

Input	Process	Output
local_computer_state left local right standby left_count local_count right_count standby_count	case local_computer_state 0,4: selected_signal ← compare_three (left, local, right, left_count, local_count, right_count) 1: selected_signal ← compare_three (standby, local, right, standby_count, local_count, right_count) 2: selected_signal ← compare_three (left, standby, right, left_count, standby_count, right_count) 3: selected_signal ← compare_ three (left, local, standby, left_count, local_count, standby_count) 5,32: selected_signal ← compare_ two (standby, right, standby_count, right_count) 6,22: selected_signal ← compare_two (standby, left, standby_count, left_count) 7,21,33,40: selected_signal ← compare_two (standby, local, standby_count, local_count)	selected_signal left_count local_count right_count standby_count

CHART D-1. DETAILED CHARTS (continued)

compare signals (continued)

```
                8,34: selected_signal ← compare_two
                      (local, right, local_count, right_count)
                   9: selected_signal ← compare_two
                      (right, left, right_count, left_count)
               10,24: selected_signal ← compare_two
                      (left, local, left_count, local_count)

          end case

     end (compare signals)
```

CHART D-1. DETAILED CHARTS (continued)

function compare_three (signal_A, signal_B, signal_C, count_A, count_B, count_C)

Input	Process	Output
signal_A signal_B signal_C count_A count_B count_C max_count (constant) tolerance (constant) past_value	calculate the following Boolean variables AB = (absolute (signal_A - signal_B) < tolerance) BC = (absolute (signal_B - signal_C) < tolerance) CA = (absolute (signal_C - signal_A) < tolerance) case AB, BC, CA true, true, true: compare_three ← median (signal_A, signal_B, signal_C) increment count_A, count_B, count_C if they are not already at max_count true, true, false: compare_three ← signal_B true, false, true: compare_three ← signal_A false, true, true: compare_three ← signal_C	count_A count_B count_C compare_three

CHART D-1. DETAILED CHARTS (continued)

compare_three (continued)

```
            true, false, false:  compare_three ← ½ (signal_A
                                 + signal_B)

                increment  count_A, count_B if they are not
                already at max_count

                decrement  count_C

            false, true, false:  compare_three ← ½ (signal_B
                                 + signal_C)

                increment  count_B and count_C if they are
                not already at max_count

                decrement  count_A

            false, false, true:  compare_three ← ½ (signal_C
                                 + signal_A)

                increment  count_C and count_A if they are
                not already at max_count

                decrement  count_B

            false, false, false:  compare_three ← past_value

                decrement  count_A, count_B, count_C

        end case

    end (compare_three)
```

CHART D-1. DETAILED CHARTS (continued)

function median (A, B, C)

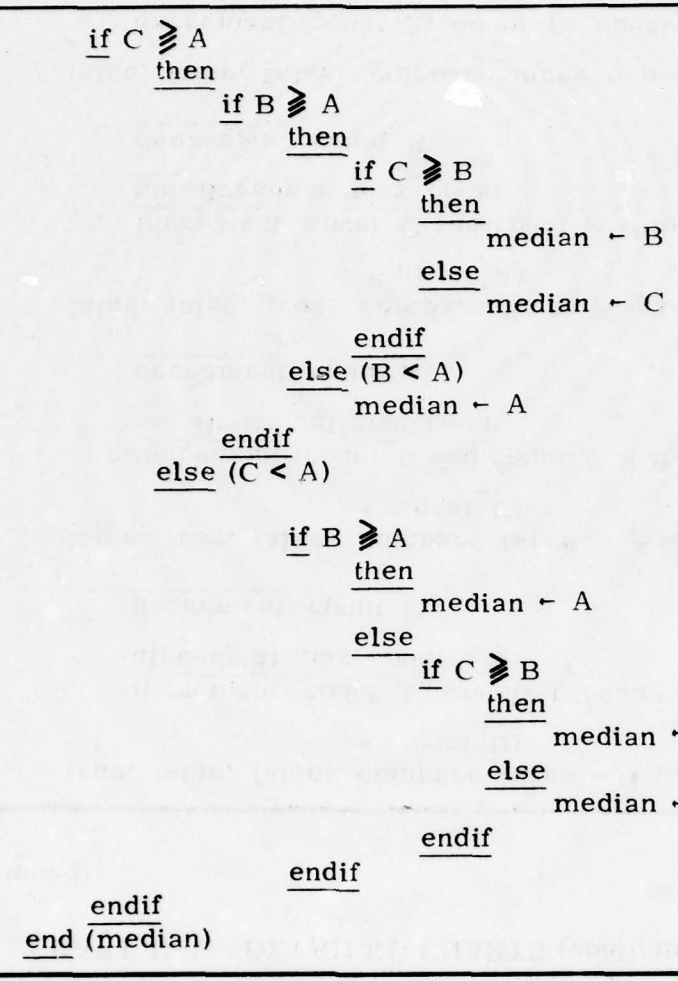
Input	Process	Output
A B C		median

CHART D-1. DETAILED CHARTS (continued)

function compare_two (signal_A, signal_B, count_A, count_B)

Input	Process	Output

signal_A
signal_B
count_A
count_B
 	
max_count
(constant)
 	
tolerance
(constant)
 	
past_value
	if absolute (signal_A - signal_B) < tolerance
	then
	compare_two ← ½ (signal_A) +
	signal_B)
	
	increment count_A and count_B
	if they are not already
	at max_count
	
	else
	
	compare_two ← past_value
	
	decrement count_A and count_B
	
	endif
	
	end (compare_two)
		count_A
		count_B
		
		compare_two

CHART D-1. DETAILED CHARTS (continued)

change state

Input	Process	Output

left_count
local_count
right_count
 	
standby_count
standby_ok
	if left_count = 0
	  then
	  left_ok ← false
	endif
	if local_count = 0
	  then
	  local_ok ← false
	endif
	if right_count = 0
	  then
	  right_ok ← false
	endif
	if standby_ok = true and standby_count = 0
	  then
	  standby_ok ← false
	endif
	if standby_ok = false and standby_count = max_count
	  then
	  standby_ok ← true
		left_ok
		local_ok
		right_ok
		
		standby_ok

CHART D-1. DETAILED CHARTS (continued)

change state (continued)

Input	Process	Output
local_computer_state	implement the state transition table of Figure 1. one way of doing this is to redefine local_computer_state as a packed word of left_computer_running right_computer_running left_ok local_ok right_ok standby_ok (this changes the numbering of the states)	local_computer_state
left_ready right_ready (external discretes)	left_computer_running ← left_ready right_computer_running ← right_ready if local_computer_state is 17, 27, 37, 47 then "stop" endif end (change state)	left_computer_running right_computer_running

CHART D-1. DETAILED CHARTS (concluded)

test standby sensor

Input	Process	Output

standby
standby_count
selected_signal
 	
tolerance
(constant)
 	
standby_factor
(constant)
 	
max_count
(constant)
	if absolute (standby - selected_signal) <
	standby_factor * tolerance
	then
	increment standby_count if it is
	not already at max_count
	else
	decrement standby_count
	endif
	end (test standby sensor)
		standby_count

APPENDIX E

PARTIAL DFCS SPECIFICATION

QUALITY ASSURANCE

A comprehensive and definitive quality assurance program shall be conducted to confirm that the design and performance requirements of Section 3 have been met. An integrated V&V methodology shall be defined at the outset of the program and shall be applied on a timely and purposeful basis.

4.1 General Requirements

The integrated V&V methodology shall be based on a complementary array of analytical, inspection, and test techniques which explicitly address all of the assurance issues relating to flight-critical embedded software. The verification activities will focus primarily on the flight software, and the validation tasks will be basically system-oriented.

Where discrepancies are detected, their nature and the resultant corrective action shall be recorded. The V&V methodology shall foster minimal repetition or revision of previously completed assurance tasks when these or any other type of software modifications are carried out.

4.1.1 Analysis

Software analysis techniques, as major elements of the V&V methodology, shall be used extensively to affirm the correctness of the flight software and to determine validation test sequences. Certain of these techniques shall be used during the design and development activities to establish the acceptability of the emerging system implementation. The scope and limitations of the analyses performed shall be noted in summarizing the associated results. Automated and self-documenting analytical techniques should be applied wherever possible.

4.1.2 Inspection

Independent reviews or walkthroughs shall be conducted on a thorough and timely basis as the design and development progress. These reviews should be performed by personnel not directly involved with the corresponding review subject, but whose backgrounds include general familiarity with the project or expertise in the technology employed. As a minimum, visual inspections of the following articles shall be conducted:

a. Specifications: system, computer program, software module

b. Design Definition: interfaces, integration, timing, software architecture

c. Test Procedures: system integration, computer program, software modules

d. Code: computer program, software modules

4.1.3 Testing

Carefully engineered testing shall be satisfactorily performed at appropriate points during the development process. These tests shall reflect and to the extent practicable establish compliance with Section 3. The details and rationale for the specific test procedures shall be based on prior software analyses, and any assumptions or limitations on the testing accomplished shall be noted and substantiated. Wherever practical, testing should be performed and evaluated on an automated basis, and the results obtained should be readily relatable to the test requirements.

4.2 Analysis Requirements

System-level requirements shall be prepared, in detail, and their approval shall be obtained from the procuring activity. Compliance with safety-related requirements shall be confirmed to the extent defined in the approved development plans prior to flight testing. Other analyses shall be completed on a timely basis to expedite system development. Analyses shall be performed on both a system and a computer program basis, and a comprehensive evaluation of the production DFCS configuration shall be accomplished.

4.2.1 System Analysis

Analysis of the system and its embedded software shall be conducted to ascertain that coincident multiple channel shutdowns cannot occur, except as a result of massive external influences.

4.2.2 Software Analysis

Analysis of the flight software shall be conducted to ascertain that its implementation contains no generic software errors or potentially critical latent software errors. This analysis shall reflect the salient details of the flight computer subsystem.

4.3 Test Requirements

System-level test requirements shall be prepared in detail, and their approval shall be obtained from the procuring activity. Compliance with safety-related requirements shall be confirmed through laboratory testing to the extent defined in the approved development plans prior to flight testing. Fulfillment of noncritical requirements may be established during concurrent laboratory and flight testing. At least the final stages of validation testing, which encompass system simulator and flight testing, shall be performed using production hardware and computer programs. Data obtained otherwise to satisfy V&V requirements shall be shown to correlate adequately with the production system test results.

4.3.1 Laboratory Tests

Although, ultimately, the requirements are to be addressed on a system level, laboratory testing shall be performed on a software, computer subsystem, and system simulator level. The series of tests shall be in accordance with the approved development plans, which reflect the integrated V&V methodology.

4.3.1.1 Software Tests

Software tests shall be performed to verify it to the extent defined in the approved development plan. This verification testing shall confirm that the software implementation is fully capable of performing as intended in the system.

4.3.1.2 Computer Subsystem Tests

Computer subsystem tests shall be performed to confirm proper hardware/software operation. This testing shall be accomplished using the flight computer properly interfaced to establish that the real-time redundancy management features are correctly implemented. It shall also establish that the subsystem performs safely under the range of admissable conditions, including multiple hardware fault conditions.

4.3.1.3 System Simulator Tests

System simulator tests shall be performed as an integral part of the validation process. Normal and faulted conditions shall be demonstrated to be within the requirements of Section 3 and the approved test requirements. Test strategies shall be designed to maximize the extent to which the flight software is exercised, and the coverage and results obtained shall be recorded through ample instrumentation and automated test capability. The test procedures shall be shown not to interfere with the intended behavior of the flight software, including that in response to inserted faults.

4.3.2 Flight Testing

After the safety of the developmental DFCS has been established the system shall be tested in operational flight conditions. The range of functional modes, simulated malfunctions, and the limits of the flight envelope shall be investigated in an efficient manner. Provisions for inflight software changes shall be such that inadvertant ones are unlikely and that unsafe ones are only remotely possible. Ultimately the overall acceptability of the production DFCS shall be confirmed.

4.4 Documentation

Documentation of the system implementation, including details of the flight software, shall be prepared and maintained through the course of the design and development process. This documentation shall be accomplished concurrently with the engineering tasks, and automated schemes should be considered. Compatibility among documentation formats and with associated V&V methods shall be established. Usefulness, understandability, and ease of updating over the life cycle of the system should be major objectives of the documentation effort.

4.4.1 FCS Development Plan

A comprehensive plan for the realization of the DFCS shall be formulated at the outset of the project and submitted for approval by the procuring activity. This plan shall incorporate the system-level aspects of the integrated V&V methodology described in the AFFDL Guidebook for DFCS software as appropriate for flight-critical functions. The plan shall detail and substantiate how

compliance with the quantitative safety and mission reliability requirements is to be confirmed, especially with regard to the flight software. Where alternative methods or other revisions to the Guidebook methodology are intended, express approval must be obtained from the procuring activity based on contractor requirements. Once the FCS development plan is approved, it shall be followed through the project.

4.4.1.1 Computer Program Development Plan

A computer program development plan (CPDP) shall be prepared which complements the FCS development plan. The CPDP shall reflect the more detailed aspects of the integrated V&V methodology and the strictly software-oriented issues in the AFFDL Guidebook for DFCS Software. Approval of the CPDP shall be obtained from the procuring activity prior to use of the plan.

4.4.2 Design and Test Data

Relevant data to support the qualification and life cycle maintenance of the DFCS shall be generated and compiled during the design and development process. These data shall include lower-level specifications, test procedures, software documentation, and comprehensive test and analysis reports.

4.4.2.1 FCS Analysis Report

The FCS analysis report shall include confirmation of compliance with the flight safety and reliability requirements. Detailed analytical results, obtained in accordance with the approved FCS development plan, shall

substantiate this compliance. Where supportive test results are also required, these shall be noted along with their significance. Analyses shall also be included where applicable to establish the basis for various validation and failure effects test details.

4.4.2.2 FCS Test Report

The FCS test report shall include confirmation of compliance with the DFCS validation requirements. Normal and faulted operation test results, obtained in accordance with approved test procedures, shall substantiate overall system acceptability. Relevant prior analyses shall be confirmed by selected test cases and procedures. Where limitations or deficiencies in testing or test results exist, explanations shall be provided which establish negligible impact on specification compliance.

APPENDIX F

PARTIAL COMPUTER PROGRAM SPECIFICATION

QUALITY ASSURANCE

4.1 Introduction

The contractor shall establish a comprehensive software quality assurance plan to verify the conformity of the CPCI to the functional and performance requirements as set forth in Section 3 of this document. The plan shall set forth specific software test plans and test procedures, analysis, and methods to be employed in verifying the computer program prepared under this specification. The plan shall also define the procedures to be followed in recording, reporting, and correcting any deficiencies revealed by or during the verification and validation processes. The plan shall be developed during CPCI design and coding phases. It will be amended as required to accommodate revisions or changes to design requirements. Detailed verification activities and methods will be selected to reflect internal CPCI structure as well as CPCI function and performance requirements.

The software quality assurance plan will be approved by the procuring agency prior to CPCI acceptance.

4.1.1 Analysis

The term "analysis method" as used herein includes any method, technique, or procedure for detecting actual and/or potential sources of anomalous or erroneous CPCI performance or behavior, with the exception of methods, techniques, and procedures as defined in Paragraphs 4.1.2 and 4.1.3. An analysis method may be partially or fully automated or manual. Analysis methods to be used in the CPCI quality assurance plan shall be as described in the following subparagraphs.

4.1.1.1 Set/Use Check

A set/use check is a systematic audit of every variable name used in a computer program, consisting of an identification of every location at which the value of the variable may be assigned or altered, an identification of every location at which the value of the variable is used, and an analysis of the precedence relationships between assignment and use.

4.1.1.2 Symbolic Evaluation

Symbolic evaluation is the process of establishing an algebraic expression relating intermediate and/or final variable names to input and/or predecessor intermediate variable names by analysis of computer program statements.

4.1.2 Inspection

As used herein, the term "inspection" is defined to mean a formalized review of the work of one or more designers/programmers in accordance with software engineering standards for structured walkthroughs.

4.1.3 Testing

As used herein, the term "testing" is defined to mean computer execution of all or part of the CPCI code for purposes of confirming CPCI behavior and the review of the results obtained from such code execution.

4.2 Analysis/Inspection Requirements

4.2.1 Analysis Requirements

The entire CPCI shall be subjected to the following analysis methods:

a. Set/Use check

b. Interface analysis

c. Flow structure analysis

This requirement shall be incorporated in the CPCI quality assurance plan.

4.2.2 Inspection Requirements

Inspection shall be used on a continuing and progressive basis throughout the CPCI development process, including detailed specification, design, module coding, and integration. This requirement shall be incorporated in the CPCI quality assurance plan.

4.3 Test Requirements

The CPCI quality assurance plan shall incorporate plans and procedures for conducting the tests specified in the following subparagraphs.

4.3.2 Category I Tests

The test requirements contained in paragraphs 4.3.2 through 4.3.4 shall constitute the Category I testing to be conducted on the CPCI.

4.3.3 Preliminary Qualification Tests

The CPCI functions/modules identified in the following subparagraphs shall be verified by preliminary qualification tests.

4.3.3.1 Initialization

Tests of the initialization function as described in paragraph 3.2.1 shall be conducted using all inputs and combinations of inputs as described in paragraph 3.2.1.1. Each input/input combination shall be in accordance with paragraph 3.1.1.2, and all output shall be in accordance with paragraph 3.2.1.3. This testing shall be accomplished prior to integration in accordance with paragraph 4.4.1.

4.3.3.2 Preflight BIT

Tests of the preflight built-in-test (BIT) function as described in paragraph 3.2.2 shall be conducted to verify the detection of all fault conditions as described in paragraph 3.2.2. This testing shall be accomplished prior to integration in accordance with paragraph 4.4.1.

4.3.4 CPCI Formal Qualification Tests

Formal qualification tests of the integrated CPCI shall be conducted in accordance with the requirements of paragraphs 4.4.2 through 4.4.4. Specific requirements shall be verified per the subparagraphs herein.

4.3.4.1 Rate-Path Structure

The rate-path structure of the CPCI shall be shown to satisfy the computational frequency requirements of paragraphs 3.2.3. Tests shall be accomplished in accordance with paragraph 4.4.2.

4.3.4.2 External Interfaces

Interface compatibility per paragraph 3.1.1.2 shall be demonstrated in accordance with paragraph 4.4.3 for inner-loop sensors and displays. Interfaces with outer-loop sensors shall be demonstrated in accordance with paragraph 4.4.4.

4.4 Test Requirements

4.4.1 Host Machine Testing

Testing conducted on a host computer may be used to demonstrate attributes and properties of algorithms which are not dependent on word length and execution time. Host machine testing may be used to validate the adequacy and correctness of algorithms which depend on Boolean or binary logic. It may be used to confirm the adequacy of numerical scaling factors for overflow

conditions provided host machine floating point features are not allowed to mask such overflow. Host machine testing shall not be used to confirm numerical resolution or freedom from underflow, unless the host machine is of the same wordlength as the target machine and is operated in fixed-point arithmetic mode.

4.4.2 Target Computer Testing

Testing conducted per this paragraph must be conducted on one or more target computers of production configuration. Interface hardware shall be sufficient to control and monitor the execution of the target computers to the degree required to accomplish the objectives of the specific test being conducted. A (non-target) computer may be used to control and monitor execution of the target computer(s), to simulate system response to target computer outputs, and to generate and/or supply inputs to the target computer.

4.4.3 Simulator Testing

Simulator testing shall be accomplished on the "iron bird" simulator defined in Paragraph 4.3.1.3 of the DFCS System Specification.

4.4.4 Flight Testing

Flight testing shall be accomplished on the flight test aircraft as identified and defined in Paragraph 4.3.2 of the DFCS System Specification.

REFERENCES

1. Bailey, D.G., "Airborne Software Structured Development," Report No. ED-DGM 4000-310 (A), Honeywell Avionics Division, Minneapolis, Minnesota, February 6, 1978.

2. Bailey, D.G., and Folkesson, K., "Software Control Procedures for the JA-37 Digital Automatic Flight Control System," AIAA Guidance and Control Meeting, San Diego, California, pp. 122-129, August 1976.

3. Bailey, D.G., and Folkesson, K., "JA-37 Digital Automatic Flight Control System (DAFCS) Self-Test Development," AGARD Report No. AG-224, Integrity of Flight Control Systems. Neuilly Sur Seine (France): North Atlantic Treaty Organization, April 1977.

4. Wirth, N., Algorithms + Data Structures = Programs. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., p. xvi, 1976.

5. Boehm, B.W.; McClean, R.K.; and Urfrig, D.B., "Some Experience with Automated Aids to the Design of Large-Scale Reliable Software," Proceedings of the International Conference on Reliable Software, Los Angeles, California, pp. 105-113, 21-23 April 1975.

6. Gerhart, S., and Yelowitz, L., "Observation of Fallibility in Applications of Modern Programming Methodologies," IEEE Transactions on Software Engineering, Vol. SE-2, No. 3, September 1976.

7. Thayer, T.A., et al., "Software Reliability Study," Final Technical Report No. RADC-TR-76-238, TRW Defense and Space Systems Group, Redondo Beach, California, August 1976.

8. Lloyd, D.K., and Lipow, M., Reliability: Management, Methods and Mathematics. Redondo Beach, California: Second edition, published by the authors, 1976.

9. Boehm, B.W., "Software and Its Impact: A Quantitative Assessment," Datamation, pp. 48-59, May 1973.

10. Boehm, B.W., Brown, J.R., and Lipow, M., "Quantitative Evaluation of Software Quality," Proceedings of the Second International Conference on Software Engineering, San Francisco, California, pp. 592-605, 13-15 October, 1976.

REFERENCES (continued)

11. Tausworthe, R.C., Standardized Development of Computer Software. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1977.

12. Yourdon, E., Techniques of Program Structure and Design. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1977.

13. Schneider, G.M.; Weingart, S.W.; and Perlman, D.M.; An Introduction to Programming and Problem Solving with Pascal. New York: John Wiley & Sons, 1978.

14. Alagic, S., and Arbib, M.A., The Design of Well-Structured and Correct Programs. New York: Springer-Verlag, 1978.

15. Lane, D., "Structured Programming Guidelines," Report 776-12882, Vol. I and II, Honeywell Avionics Division, St. Petersburg, Florida, September 1977.

16. Reifer, D.J., and Trattner, S., "A Glossary of Software Tools and Techniques," Computer, Vol. 10, No. 7, pp. 52-61, July 1977.

17. Benenati, C.J., and van Dam, A., "An Informal Survey of Software Engineering Tools and Methodologies," Raytheon Submarine Signal Division, Providence, Rhode Island, December 7, 1977.

18. Miller, E.F., "Program Testing Tools: A Survey," presented at MIDCON, Chicago, Illinois, November 8, 1977.

19. Schlicht, R.A.; Brueggeman, W.M.; and Staley, G.W.; "1975 ADG Software Program: Verification/Validation," Vol. I-V, Honeywell Systems and Research Center, Minneapolis, Minnesota, 1975.

20. Smith, R.L., "Validation and Verification Study," from Structured Programming Series, Vol. XV. Gaithersburg, Maryland: International Business Machine Corporation, May 22, 1975.

21. Furtaw, R.N.; Edson, D.T.; Teutsch, R.P.; and Steiner, J.S.; "Aircraft Avionics Tradeoff Study," Report No. ASD/XR 73-19, Hughes Aircraft Co., Canoga Park, California, pp. 307-369, November 1973.

22. Ramamoorthy, C.V., and Ho, S.F., "Testing Large Software with Automated Software Evaluation Systems," IEEE Transactions on Software Engineering, Vol. SE-1, No. 1, March 1975, pp. 46-58. Also in R.T. Yeh (ed.), Current Trends in Programming Methodology, Vol. 2. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., pp. 112-150, 1977.

23. DeWolf, J.B., and Wexler, J., "Approaches to Software Verification with Emphasis on Real-Time Applications," AIAA Computers in Aerospace Conference, Los Angeles, California, pp. 41-51, October 31 to November 2, 1977.

24. Hartwick, R.D., "Test Planning," AFIPS Conference Proceedings, Vol. 46, pp. 285-294, 1977 National Computer Conference.

25. Fairley, R.E., "Tutorial: Static Analysis and Dynamic Testing of Computer Software," Computer, Vol. 11, No. 4, pp. 14-23, April 1978.

26. Huang, J.C., "Program Instrumentation and Software Testing," Computer, Vol. 11, No. 4, pp. 25-32, April 1978.

27. Panzl, D.J., "Automatic Software Test Drivers," Computer, Vol. 11, No. 4, pp. 44-50, April 1978.

28. Darringer, J.A., and King, J.C., "Applications of Symbolic Execution to Program Testing," Computer, Vol. 11, No. 4, pp. 51-60, April 1978.

29. Elspas, B.; Levitt, K.N.; Waldinger, R.J.; and Waksman, A.; "An Assessment of Techniques for Proving Program Correctness," ACM Computing Surveys, Vol. 4, No. 2, June 1972.

30. Hantler, S.L., and King, J.C., "An Introduction to Proving the Correctness of Programs," ACM Computing Surveys, Vol. 8, No. 3, pp. 331-353, September 1976.

31. London, R.L., "Perspectives on Program Verification," in R.T. Yeh (ed.), Current Trends in Programming Methodology, Vol. 2. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., pp. 151-172, 1977.

32. Maurer, W.D., "Proving the Correctness of a Flight-Director Program for an Airbourne Minicomputer," SIGMINI/SIGPLAN Meeting Proceedings and SIGPLAN Notices, Vol. 11, No. 4, pp. 103-108, April 1976.

33. Robinson, L., and Levitt, K.N., "Proof Techniques for Hierarchically Structured Programs," in R.T. Yeh (ed.), Current Trends in Programming Methodology, Vol. 2. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., pp. 173-196, 1977.

34. Boebert, W.E.; Kamrad, J.M.; and Rang, E.R.; "The Analytic Validation of Flight Software: A Case Study," NAECON 1978, Vol. 1, pp. 242-248, May 26, 1978.

35. Peterson, J.L., "Petri Nets," ACM Computing Surveys, Vol. 9, No. 3, pp. 223-252, September 1977.

36. Babich, A.F., "Proving the Correctness of Parallel Programs," IEEE Computer Society, Paper R-78-21, February 1978.

37. Saib, S.H.; Benson, J.P.; and Melton, R.A.; "Software Quality Laboratory," presented at AIAA Computers in Aerospace Conference, Los Angeles, California, October 31 to November 2, 1977.

38. Straeter, T.A.; Foudriat, E.C.; and Will, R.W., "MUST: An Integrated System of Support Tools for Research Flight Software Engineering," Paper no. 77-1459, AIAA Computers in Aerospace Conference, Los Angeles, California, pp. 442-446, October 31 to November 2, 1977.

39. Goodenough, J.B., and Gerhart, S.L., "Toward a Theory of Test Data Selection," IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, June 1975, pp. 156-173. (Also revised in R.T. Yeh (ed.), Current Trends in Programming Methodology, Vol. 2. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., pp. 44-79, 1977.

40. Denning, P.J.; Dennis, J.B.; and Qualitz, J.E.; Machines, Languages, and Computation. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1978.

41. Katzan, H. (Jr.), Systems Design and Documentation: An Introduction to the HIPO Method. New York: Van Nostrand-Reinhold, 1976.

42. Roubine, O., and Robinson, L., "SPECIAL Reference Manual," Third Edition, Report No. CSG-45, Stanford Research Institute, Menlo Park, California, January 1977.

43. Dyer, M.C., "Design and Development of Distributed Systems for Aerospace: A Hardware/Software Approach," Paper no. 77-1450, AIAA Computers in Aerospace Conference, Los Angeles, California, pp. 387-393, October 31 to November 2, 1977.

44. Wensley, J.H., et al., "Design Study of Software-Implemented Fault-Tolerant (SIFT) Computer," SRI International, Menlo Park, California, June 1978.

45. Enslow, P.H., "What is a 'Distributed' Data Processing System?," Computer, Vol. 11, No. 1, pp. 13-21, January 1978.

46. Jensen, E.D., "The Honeywell Experimental Distributed Processor--An Overview," Computer, Vol. 11, No. 1, pp. 26-36, January 1978.

47. Enslow, P.H., "Multiprocessor Organization--A Survey," ACM Computing Surveys, Vol. 9, No. 1, pp. 103-129, March 1977.

48. Wise, C., and Bain, J., "Distributed Processing for MIL-STD-1533A," ASD-TR-78-34, Proceeding of the Second AFSC Multiplex Data Bus Conference, Dayton, Ohio, 1978.

49. Katt, D.R., and Raymont, P.A., "The Flight Control Computers of the F-18 Electronics--Flight Control," Second Digital Avionics System Conference, Los Angeles, California, November 2-4, 1977.

50. Myers, G.J., "A Controlled Experiment in Program Testing and Code Walkthrough/Inspections," Communications of the ACM, Vol. 21, No. 9, pp. 760-768, September 1978.

51. Boyd, D.; Pizzarello, A.; and Vestal, S.C.; "Rational Design Methodology," Report No. HR-78-257:17-38, Honeywell Corporate Technology Center, Minneapolis, Minnesota, June 1978.

52. Gannon, C., "A Verification Case Study," Paper no. 77-1443, AIAA Computers in Aerospace Conference, Los Angeles, California, pp. 349-369, October 31 to November 2, 1977.

53. Godoy, S.G., and Engels, G.J., "Software Sneak Analysis," Paper no. 77-1386, AIAA Computers in Aerospace Conference, Los Angeles, California, pp. 63-87, October 31 to November 2, 1977.

54. Clarke, L.A., "A System to Generate Test Data and Symbolically Execute Programs," IEE Transactions on Software Engineering, Vol. SE-2, No. 3, pp. 215-222, September 1976.

55. Fisher, D.A., "The Common Programming Language Effort of the Department of Defense," 1977 Computers in Aerospace Conference, Los Angeles, California, pp. 297-307, November 1977.

56. Heninger, K.L., "Specifying Software Requirements for Complex Systems," Specification of Reliable Software, Cambridge, Massachusetts, pp. 1-14, April 1979.

57. Heninger, K.L.; Kallander, J.W.; Stone, J.E.; Parnas, D.L.; "Software Requirements for the A-7E Aircraft, NRL Memorandum Report 3876, Naval Research Laboratory, Washington, D.C., November 1978.

*U.S.Government Printing Office: 1979 — 657-002/123